November 1985

UILU-ENG-85-2232
CSG-48

COORDINATED SCIENCE LABORATORY
College of Engineering

A PARALLEL STACK PROCESSOR TO REDUCE PROCEDURE-CALL OVERHEAD

Richard James Eickemeyer

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-85-2232 (CSG-48)	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Naval Electronic Systems Command VHSIC Joint Services Electronic Program
6c. ADDRESS (City, State and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code) Arlington, VA 22202
8a. NAME OF FUNDING/SPONSORING ORGANIZATIONS Naval Electronics Systems Command VHSIC Joint Services Electronic Progr. 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00039-80-C-0556 (VHSIC) N00014-84-C-0149 (JSEP)
8c. ADDRESS (City, State and ZIP Code) Arlington, VA 22202	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: N/A; PROJECT NO.: N/A; TASK NO.: N/A; WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) A Parallel Stack Processor to Reduce Procedure-Call Overhead	
12. PERSONAL AUTHOR(S) Richard James Eickemeyer	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) November, 1985 15. PAGE COUNT 84
16. SUPPLEMENTARY NOTATION None	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) computer architecture, multiple register sets, performance, procedure calls, RISC, VLSI processor

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A processor organization is presented to reduce the large overhead of procedure calls in high-level languages. In the Parallel Stack Processor (PSP), processor registers are each at the top of a hardware stack of registers. Saving processor registers on a procedure call takes place in one cycle by pushing all registers simultaneously. A detailed performance model, driven by dynamic high-level language statistics, is presented. Results from the model indicate the effect on performance of the parallel stack architecture when compared to a processor without parallel stacks. THe processor architecture is specified in the report along with a discussion of implementation details for the VLSI single-chip processor.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code) 22c. OFFICE SYMBOL None

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

A PARALLEL STACK PROCESSOR TO REDUCE PROCEDURE-CALL OVERHEAD

BY

RICHARD JAMES EICKEMEYER

B.S., Purdue University, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Motivation

With the encouragement of structured programming, the procedure call has become a very frequent operation in a computer program. Several studies have shown that procedure calls make up over ten percent of executed instructions in High-Level Language (HLL) programs [PaSe82, Tane78, Weic84]. Because of the writing and debugging advantages of structured programming, one cannot expect these figures to decrease in the future. Each procedure call, in turn, makes several references to memory in saving registers and passing parameters. Patterson and Séquin [PaSe82] state that approximately 45 percent of all memory references are in the handling of procedure calls and returns. Because memory references require off-chip access, they are slow. Clearly, a reduction of the number of memory references could lead to significant improvements in overall processor performance.

The typical method of calling another procedure makes use of a stack in memory; a block of memory, a *call frame*, or *activation record*, is allocated at each procedure call. For example, consider the calling method of the VAX 11/780 using the CALLS instruction [LeEc80]. First, parameters are pushed onto the stack. Pointers are pushed for *call-by-reference* parameters. Next, some registers are saved on the stack based on a mask provided by the called procedure. Various pointers and other state information are then put onto the stack. Finally, local variables can be allocated on top of the stack. Within the procedure, access to parameters and local variables requires memory references. To return to the calling procedure, the processor's previous state must be reconstructed from the information on the stack. From this, one can see why so many memory

references need to be made for each procedure call.

1.2. Previous Research

The Berkeley Reduced Instruction Set Computer (RISC) [PaSe82, Fitz82] has a small instruction set which keeps the processor design simple and, consequently, reduces the chip area needed for control. The extra space on the chip is used for storage organized as a number of register sets, or windows. When a procedure is called, a new register set is allocated. The register set comes from an on-chip register file where a register window pointer indicates the location of the current window in the RAM. The windows overlap, allowing parameters to be passed in some registers from the calling procedure to the called procedure. A procedure call involves changing the window pointer, handling overflow, and transferring to the called procedure. Each register in the register file is given an address, so that registers not in the current window can be accessed. Lampson [Lamp82] also proposes using multiple register banks. Sites [Site79] compares different methods of using many registers. The effects and costs of multiple register sets, in general, are also being studied [Colw83].

In the IBM 801 [Radi83], also a reduced instruction set computer, a different approach is used. By expanding some procedures in-line, there are fewer procedure calls. Advanced compiler technology [Chai82] optimizes the use of registers for the processor's single register set of size 32. Dannenberg [Dann79] also proposes a single large register set for local variables.

The Bell Labs C Machine [DiMc82, Hill83] uses a number of registers organized as a circular buffer for the top of the stack. There are 1024 32-bit registers. The key to this machine is that some instructions can be partially decoded when they are fetched from memory and placed in the instruction cache. This decoding changes stack pointer offsets into addresses and those addresses that are in registers are changed to register references. When the instruction is executed, most references will be as efficient as register accesses. The registers are invisible to the compiler. When a procedure is called, space is allocated in the registers. If there is not enough room, some registers

are flushed to memory.

1.3. Parallel Stack Processor

The Parallel Stack Processor (PSP) also uses a number of register banks. The organization is somewhat different from those previously mentioned. In this case, storage is organized as a number of stacks, operating in parallel. There is one stack for each processor register. The stacks are implemented as shift registers, so that data are physically moved up and down the stack, unlike most other stacks in RAM where the data are stationary but pointers are updated. Each level in the stacks corresponds to a different procedure. Like RISC, but unlike the C Machine, a fixed number of registers is available for each procedure. A difference from some of the methods mentioned above is that only the top of the stacks, the processor registers, can be addressed directly. This allows a large amount of on-chip storage to be available while using a small address to access the data. In addition, the processor's registers are always in the same physical locations so register references do not have to be converted to RAM addresses. A small address results in small decoders and faster decode time, which keeps the length of an instruction cycle short. Data at lower levels of the stacks cannot be accessed until brought to the top level as part of a procedure being executed. Because of the method of stack access, the number of stack levels is completely independent of the rest of the processor—any number of stack levels can be put on a chip, assuming enough chip area is available, and no changes need be made to the remainder of the design. In comparison with partial window overlap in RISC, the PSP provides full overlap of registers between called and calling procedures. The parallel stack architecture is presented in Chapter 2. Some comments on the use of the stacks for HLL procedure calls are also given.

Two methods of using the stacks are considered. One method is to use the registers for variable allocation and for parameter passing. Parameters can be passed in registers because the register value is not changed when the stacks are pushed for a procedure call. On a return, data are brought up one level by popping the stacks. Local variables that were allocated in registers are

overwritten on a pop. The second method of using the stacks is a traditional allocation and parameter passing algorithm with the parallel stacks used only to save registers. Ideally, a procedure call can be executed in one CPU cycle, by simultaneously pushing the registers on stacks. However, the performance is degraded by stack overflow, lack of sufficient space for a procedure's parameters or variables, and rearrangement of parameters. In Chapter 3, a detailed performance model is described for PSP which takes these degradations into account. The model is based on HLL statistics and models not only procedure calls, but all HLL statements in order to measure overall processor performance. When compared to a conventional processor architecture, the PSP shows a significant performance improvement.

A processor is proposed which incorporates the parallel stack architecture. The organization of PSP is described in Chapter 4. PSP fits into the reduced instruction set category with RISC and the 801. There is a small number of instructions and, except for a few instructions, all instructions take the same amount of time to execute. This allows for a simple processor design which saves chip area and design time. With few instructions, instruction decoding is fast, so the cycle time can be kept short. There are also benefits for compiler writers in that the compiler does not have to choose the best of several possible sequences of instructions. The advantages and disadvantages of reduced instruction sets are not discussed further in this thesis, but can be found in the works of several others [ClSt80, Colw83, DiPa80, PaDi80, Patt85]. The PSP instruction set is presented in Chapter 5.

Design issues of the PSP are described in Chapter 6. This discussion includes detailed instruction timing specifications. Concluding remarks are in Chapter 7.

CHAPTER 2

PARALLEL STACK ARCHITECTURE

2.1. Introduction

In this chapter, a method to decrease procedure-call overhead is presented. The parallel stack architecture consists of a number of on-chip stacks which operate in parallel. The top of each stack is accessible as a processor register. Registers are saved on a procedure call in one cycle by pushing all stacks simultaneously. Access is allowed to the bottom of the stacks from a processor bus to handle overflow. There is no access to internal levels of the stacks. The stack mechanism is well suited for procedure parameter passing and local variable allocation. In this chapter, the parallel stack architecture is presented. Some methods of using the stacks during program execution are discussed.

2.2. Parallel Stack Operation

2.2.1. Basic Stack Operation

The parallel stack architecture is shown in Fig. 2.1. During normal processing, the Top of Stack Registers (TOSRs) are the processor's registers. When a procedure call occurs, all the stacks are pushed so that the TOSR contents are copied to the stack level below the top. At the same time, the contents of each stack level are moved to the next lower level. The contents of the TOSRs are not altered by a stack push. This allows the calling procedure to leave data in the TOSRs for the called procedure. The Program Counter (PC) of the calling procedure, the return address, must be saved. The PC, together with some other status information, such as the interrupt mask, is contained in the Program Status Word (PSW). The PSW is given its own stack. READ access is

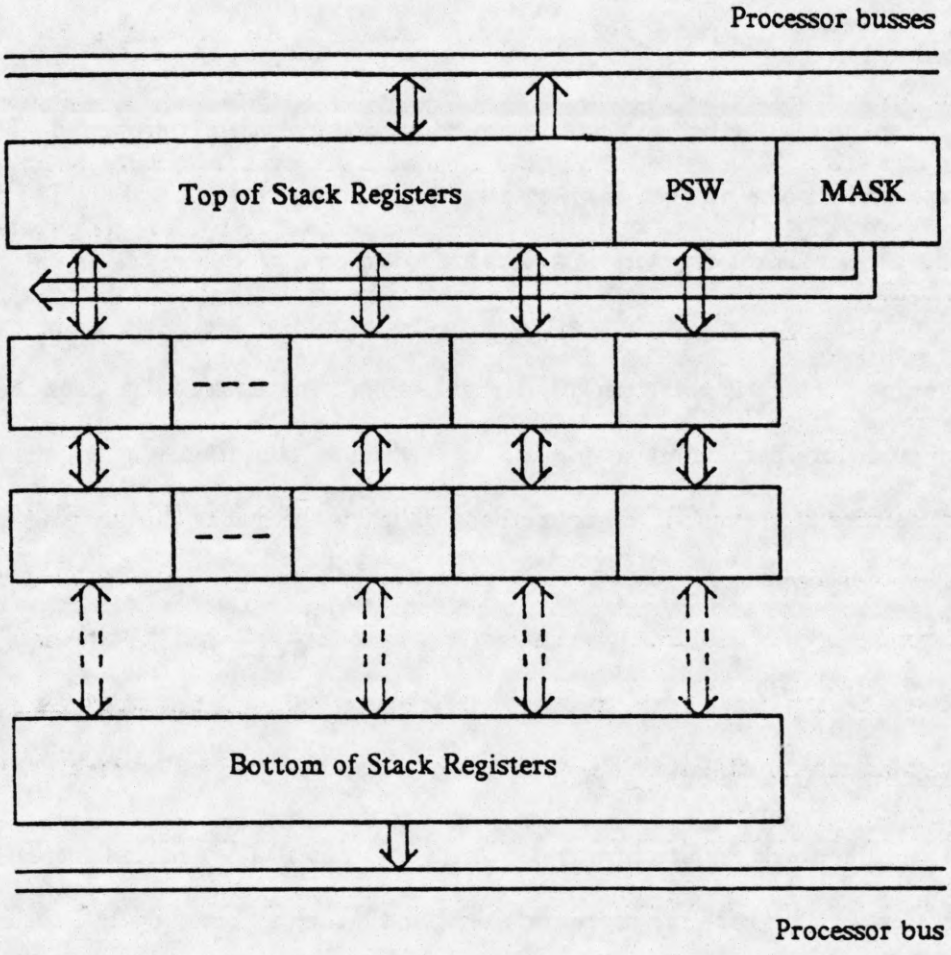

Fig. 2.1. Register stack organization.

provided to the Bottom of Stack Registers (BOSRs). This allows one to remove data from the stacks should they become full.

When returning from a called procedure, all the stacks are popped. This moves the data at each level of the stacks up one level. The TOSRs are overwritten in this process. As the PSW is also popped, the return address is made available and processing can continue from the point where the procedure was called. With one exception, when stacks are pushed or popped all the stacks are pushed or popped. The PSW stack is allowed to operate independently of the other register stacks on some occasions to be described later. During a normal CALL, however, the PSW stack is pushed with the other stacks. In addition to the stack registers, the PSP has a set of registers not associated with stacks (not shown in Fig. 2.1). These Global Registers (GRs) are identical to registers on a conventional processor and can be used for global or temporary data.

2.2.2. The Return Mask

While the stacks all operate together, there is a simple way to allow some independence between the stacks. A mask (see Fig. 2.1) is used on a RETURN to prevent the data in specified TOSRs from being overwritten. When a particular register is "masked," the data that were at the level below the top are lost when the stacks are popped. If the register is not masked, it is overwritten as described above. The use of the mask allows the called procedure to leave data in the TOSRs for the calling procedure when it resumes. The mask is specified in the RETURN instruction and consists of one bit corresponding to each of the TOSRs and PSW.

2.2.3. Overflow and Underflow

As procedures call other procedures, there could come a time when the stacks are full. The depth of the stacks must be sufficient to hold register contents of several levels of procedure nesting. There are cases, such as recursive calls, where the depth of the nesting could be very large. This happens for any sequence of CALLs and RETURNs where the number of CALLs exceeds the number of RETURNs by a number greater than the depth of the stacks. Because only a limited

number of stack levels can be provided, there must be a method for handling deep nesting. When the stacks contain data at all levels and a CALL instruction is pending, a stack *overflow* occurs. Executing the CALL would cause data at the bottom to be lost. By allowing access to the bottom element of each stack, however, the stack can be continued in memory. Because the parallel stack architecture is designed to reduce memory references during a CALL, the frequency of stack overflows must be small. This implies that the stack depth must be large enough to handle most sequences of CALLs and RETURNs. A CALL instruction must include a check for overflow. If no overflow would result, the stacks are pushed, saving the processor state. The address of the called procedure is loaded into the PC. If an overflow occurs, the BOSRs must first be written to memory and then the CALL can proceed as in the no overflow case.

Once some data have overflowed and have been written to memory, there will come a time when they should be brought back to the processor. One method of doing this is to fill the bottom of the stacks as soon as they are free. Doing this, however, will increase the number of overflows if a CALL occurs shortly thereafter. The method which most reduces the overflow rate is bringing the overflow data in only when they are needed by a procedure currently being executed. A stack *underflow* is defined as the state when the TOSR level is the only level of the stacks with valid data and a RETURN is to be executed. Executing the RETURN would cause garbage to be popped into the TOSRs. A RETURN instruction, therefore, must check for underflow. If there is none, the stacks are popped. If there is an underflow, data from memory are brought into the TOSRs directly. READ access is required of the BOSRs to handle overflow, but WRITE access is not required (see Fig. 2.1). A Stack Level Indicator (SLI), indicating what stack levels have valid data, is needed to generate the overflow and underflow signals.

A method for moving stack overflow and underflow data to and from memory must be considered. A stack should be implemented in memory to keep the overflow data. The memory stack management could be done in hardware or software. A hardware approach requires a special regis-

ter for the memory stack pointer. BOSRs would be stored sequentially and the pointer would be updated. On a return, the TOSRs could be loaded in a similar manner. This method would be fast because there would be few, if any, instruction fetches required which would compete for memory with data movement. The chip design would be complicated by this extra logic. The speed gained by the increased complexity could be gained just as easily by increasing the stack depth to reduce the number of overflows. While this also adds area to the chip, it is done by simply repeating an already designed stack level element.

Instead of hardware, software routines are used to handle the overflow and underflow. The only requirement of the additional hardware is to recognize the overflow and supply an address where execution should continue. Because the PC is changed when executing the overflow routine, the old PC must be saved. The stacks are full, however, so where is this PC stored? One option is to allow the PSW stack to operate independently of the other stacks. Extra levels in the PSW stack would be needed. A second option is to have a separate register to save the PSW on overflows. The latter adds unneeded complexity to the chip as a new port to the PSW is needed. Adding extra levels to the PSW stack is easy. The additional control logic needed for an independent stack is not very complicated.

The method of handling overflow and underflow can now be described in more detail. The PSW stack contains two levels more than the other stacks. It can be pushed and popped independently of the other stacks, but this is only necessary for handling overflow and underflow; hence, the SLI keeps track only of the register stack level, not the PSW stack level. If the user were allowed to control the stacks independently in general usage, two types of overflow and underflow would have to be checked. On a normal CALL or RETURN all stacks are pushed or popped together. When an overflow is recognized, the PSW stack is used to save first the current address and then the called procedure's address. The processor provides an address for the overflow routine. This routine stores each BOSR in memory and transfers to the called procedure by popping

the PSW stack.

For each overflow or underflow, the data of only one stack level are moved to or from memory. Because data are loaded directly into the TOSRs on underflow, restoring multiple levels at each underflow is not possible. Saving multiple stack levels on a CALL overflow is not possible in the PSP, although it could be made possible with increased access to the SLI. It can be shown, however, that moving one level at a time is the best policy for managing multiple register sets when one is not able to predict future stack operations. Moving one level at a time (ignoring the overhead for entering and leaving the overflow/underflow routines) provides the minimum number of transactions between the register stacks and the memory. Any other method can offer no fewer transactions.

2.2.4. Interrupts and the Stacks

In addition to saving the processor state on procedure calls, interrupts can use the stacks for the same purpose. Once an interrupt has occurred, it is identical to a CALL except the address is specified by the processor instead of the program. Overflow is handled the same as when a CALL generates an overflow.

2.2.5. Implementation

A straightforward implementation of the architecture described above is with shift registers. One register stack consists of a number of one-bit shift registers operating in parallel. This implementation is assumed throughout this paper and is described in more detail in subsequent chapters. The important points of this implementation are that only a small number of registers can be addressed directly, resulting in small decoders. It is also very easy to change the stack depth in a design, again because intermediate stack levels can be accessed only by their neighbors.

Other implementations of the parallel stack architecture may be possible, however. One such implementation uses RAMs instead of shift registers. The TOSRs are the same as in the shift-

register implementation, but on a procedure call the TOSRs are copied, in parallel, to individual RAMs. This requires an additional bus for each register. A pointer is needed to indicate the tops of the stacks in RAM and another to indicate the bottoms. Overflow and underflow are determined by comparing the pointers. Data can be removed from the bottom of the stacks by specifying a particular BOSR. The BOSR number specifies which RAM and the bottom of stack pointer selects the element in the RAM. By allowing access to arbitrary stack levels, individual internal stack elements can be addressed, which is not possible in the shift-register implementation. This could be used for up-level addressing in a manner similar to that used in RISC [Kate83]. Because TOSR data are copied on a CALL, the return mask can be used on a RETURN exactly as in the shift-register implementation.

Another implementation using RAM is to have one RAM for all the stacks. Registers are saved by copying their values one at a time to this RAM. However, each TOSR has a BOSR (stack depth 2) from which copying occurs. This allows execution to continue using TOSRs while the registers are copied from BOSRs. The BOSRs contain the data to be popped on a RETURN. Sites [Site79] refers to this as *dribble-back*; however, he considers moving the data to memory rather than to an on-chip RAM. The performance of this implementation would suffer when there were CALLs occurring close together or, similarly, for RETURNs executed in close proximity. This penalty could happen on up to 30% of the CALLs and RETURNs [Kate83]. This time could be reduced by pipelining the data movement and achieving more than one transfer per CPU cycle. The return mask and up-level addressing can be used with this technique. The copying mechanism adds complexity to the design since there is a possibility of execution waiting for copying and the BOSRs must be kept informed of up-level addressing changes made in the RAM. This implementation removes the storage from the data path, which could result in a faster cycle time due to shorter busses. However, an additional bus is required, as in the individual RAM implementation.

2.3. Programming Considerations

2.3.1. Conventional Procedure Call and Return

In a conventional processor with a conventional compiler, a procedure call requires several memory references. A stack is implemented in memory where space is allocated for procedure related data. (See Aho and Ullman [AhU177] for some general information about calling sequences and Levy and Eckhouse [LeEc80] for an example of the VAX 11/780 calling sequence.) When a procedure is called, new space is allocated on the top of the stack. A typical example of this *call frame* is shown in Fig. 2.2. A calling procedure begins the new frame by moving its actual parameters, the called procedure's formal parameters, to the top of the stack. The two common methods for passing parameters are *call-by-value* and *call-by-reference*. The called procedure is allowed to use the locations in which parameters were passed for its own purposes. However, the calling procedure's copy of a *call-by-value* parameter is not allowed to be changed by the called procedure. For *call-by-value*, therefore, a copy of the parameter is made and put on the stack. The calling procedure's value of a *call-by-reference* parameter can be changed. This is a frequent method to

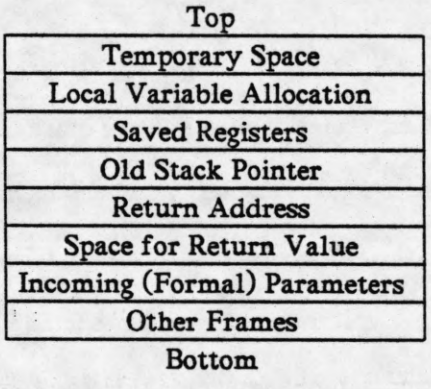

Fig. 2.2. A typical call frame.

return multiple values to the calling procedure. The common method of passing these parameters is to pass the address instead of the value. By giving the called procedure a variable's address, it can change the calling procedure's copy of this variable. The calling procedure may also leave some space on the stack for a return value.

After parameters have been passed, the current processor state must be saved and the new procedure must be called. Processor registers are saved on the stack as well as the return address and one or more stack pointers. Not all the registers may need to be saved. In the VAX, for example, the called procedure begins with a mask specifying which registers should be saved. When a procedure has been called, the stack is used to allocate space for local variables and temporary storage. If this procedure calls another, it places the actual parameters on the top of the stacks beginning a new *call frame*.

When the called procedure has completed execution, it returns to the called procedure. Space for formal parameters, temporaries, and local variables is no longer needed. The processor state is restored by loading registers and stack pointers. The called procedure is resumed when the return address is loaded into the PC. A RETURN is faster than a CALL because a CALL needs to put data in most of the locations in the *call frame* while a RETURN can skip over many of those locations.

2.3.2. PSP Procedure Call and Return

The PSP could be used in a way similar to that described above. A *call frame* could be set up in memory for parameter passing and local variable allocation. Processor state, however, can be saved in the register stacks. This would greatly reduce the number of memory references for a CALL. A RETURN would require no memory references.

The above method of using the PSP is simple but fails to take advantage of one of the PSP's features—the return mask. The mask allows variables to be allocated and parameters to be passed in registers. The value of a *call-by-value* parameter is passed, as in the conventional processor, but it is passed in a register. The called procedure can change this copy, but on the RETURN, when the

register is not masked, the copy is lost and the old value is restored from the register stack. Because the mask allows values to be returned to the calling procedure, the value of *call-by-reference* parameters can also be passed in registers. In this case, the mask specifies that the data in the TOSR should not be overwritten on a pop. The calling procedure's value is lost, and the new value from the called procedure is left in the register. A function, a procedure that returns a value, can use the mask by reserving one register for a return value and treating it like a *call-by-reference* parameter when returning.

The mask also allows variables to be allocated in registers. In a conventional processor this could be a problem if the variable is used as a *call-by-reference* parameter, because registers do not have addresses. In the PSP, an address is not needed for *call-by-reference* parameters. Because the address is passed in a conventional processor, references to the formal parameter require a level of indirection. The same reference in the PSP can be made directly to the value in a register. Temporary data used in expression evaluation can be kept in the GRs. Because these data are short-lived and do not extend across procedure calls, those registers will not be in use when a procedure call occurs and hence, do not have to be saved. The other use of the GRs is for global data, i.e., data that could be used by any procedure. Because they are the same for all procedures, these registers do not have to be saved on a procedure call.

The above method of passing *call-by-reference* parameters is called *copy-restore*. Using this method, several copies of a variable may exist. For example, this occurs if the same actual parameter is passed to more than one formal parameter as *call-by-reference*, or if a global variable is passed to a procedure as *call-by-reference*. This aliasing of a variable causes changes in one copy of a variable but not in the other aliases; therefore, several copies of the same variable will have inconsistent values. One solution to this problem is to use the traditional method for passing this type of parameter. Registers can still be used, but the address of the parameter must be passed. A solution that permits *copy-restore* is to allow the programmer to specify in the source code which

procedures can use *copy-restore* and which cannot.

Although the use of registers eliminates most of the need for a *call frame* stack in memory, it cannot be completely eliminated. There may be procedures with a large number of parameters or local variables so that the number of registers is not sufficient to hold them all. In addition to the size problem, array indexing requires addresses. Local arrays and other structures are allocated in the memory *call frame*. The *frame pointer* (FP), which points to the current procedure's *call frame* in the *call frame stack*, can be kept in a register stack, so that referencing data in memory with an FP offset can be done quickly. This also allows easy saving and restoring of the FP at CALL and RETURN.

When the compiler encounters a CALL instruction, code must be generated to build the *call frame* and move the parameters to the correct locations. The procedure to be called provides some information on how to build the *call frame*. It must specify whether registers or memory is to be used for each part of the *call frame*. The location of each formal parameter must be the same for each call to a specific procedure. The easiest way to do this is to use the order of parameters in the procedure heading and a convention establishing register usage. An example of this is to pass parameters in order, in sequential register order. If there are not enough registers, memory is used for the remaining parameters. Parameter passing consists of moving the actual parameters to the proper location. The value of scalars passed in registers is passed regardless of whether the parameter is *call-by-value* or *call-by-reference*. Scalars that need to be passed in memory are passed in the conventional manner. Addresses are used when passing arrays or other structures. There are instances, such as a register-allocated variable which is to be a *call-by-reference* parameter passed as an address in memory, where some data movement not found in a conventional processor is needed in the PSP. In this example, the variable would first be moved to memory so it could be given an address. After the RETURN, the new value may be copied back to the register. Another example is register-allocated variables not used as parameters which must be saved in memory to make

room for actual parameters.

In summary, parameters are moved to the proper location. A CALL instruction is issued which saves the processor state when the stacks are pushed. This includes the PSW and the FP. In the called procedure, local variables are allocated in unused registers or in memory. To return to a calling procedure, a RETURN instruction deallocates the local variable space, restores or changes the value of parameters, and restores the PSW and FP of the calling procedure. After the RETURN, some additional data movement may be required, especially among the registers, to get variables to the proper location to continue program execution.

When a CALL causes an overflow, data are moved to memory. There are a few possible options to choose where in memory the overflow data should go. The simplest, and the one assumed in the remainder of this paper, is to have a separate stack for overflow data. This method is simple because each stack frame is of a fixed size so no pointers are needed to link between frames. One pointer to the top of the stack is needed. Another option is to allocate space in each *call frame* so that in the event of an overflow of the data associated with the *call frame* there is space reserved for the overflow data. This requires modifying the calling sequence to include leaving the extra space. Additional bookkeeping may be required as the active procedure will be distant from the overflowing procedure. A third option is to allocate space on the heap. This requires pointers to link the overflow frames and an interface to heap allocation and deallocation routines.

2.3.3. High-Level Language Use of the PSP

The Parallel Stack Processor is intended to be programmed in High-Level Languages. A few comments on this aspect of the machine are necessary. In a block structured language, one can declare a variable local to a block, but can reference that variable from any procedure that is defined within the same block. Because there is no access to the register stacks except from the top, a variable defined as above, could not be accessed if it were allocated in a register and then pushed to an internal level of the stacks. In Pascal [JeWi74], for example, because statically nested

procedures are compiled before the enclosing procedure, the compiler could check which variables are referenced by other procedures. These variables would have to be allocated in memory in the *call frame*. Those variables not referenced by other procedures could be allocated in registers or memory as usual. A procedure that calls no other procedures or only calls itself knows that its local variables cannot be accessed by other procedures. In Pascal, one specifies in the procedure heading whether each parameter is *call-by-value* or *call-by-reference*. This is easily taken advantage of in the parallel stack architecture by using the mask.

The C language [KeRi78] is not a true block structured language such as Pascal or Algol. All procedures in C are at the highest level and one cannot reference variables of one procedure from any other. This is consistent with the hardware stack, since the only access to the stack is from the top of the stack. In C, all parameters are *call-by-value*. *Call-by-reference* is achieved by passing pointers to variables. It is these pointers which normally cause problems in a register machine. If a pointer to a scalar is used as an actual parameter, then it is reasonable to assume that the pointer is used to produce a *call-by-reference* parameter and that no address arithmetic will occur within the called procedure. Arrays and other data structures should be passed as pointers.

2.3.4. Context Switch

In a multiprogramming environment, to execute a context switch, the state of the current process must be saved and the state of the new process must be brought in. A context switch in the PSP may seem slow due to a large context in the parallel stacks. However, in most operating systems the time to save registers is only a small fraction of the overhead involved in the processing of a context switch.

To execute a context switch, the operating system takes control by interrupting the current process. This causes the TOSRs and PSW of the interrupted process to be pushed. Emptying the stacks is done by pushing the stacks using a string of CALLs. Those levels of stacks that are empty are pushed out before any overflow occurs. Those levels that cause overflow are written to

memory. The GRs are then saved on top of the overflow stack. Finally, any other information in memory that might have to be moved is moved.

When the next process is to be brought in, stack pointers are restored and GRs are loaded. A RETURN instruction is issued causing an underflow. This results in the TOSRs and PSW being loaded and control transferring to the new process. Only one level of the stacks is restored after a process swap. Other levels will be brought in, as needed, with RETURN underflows. The only special feature needed for process swaps is that the SLI must be cleared once the first process has been saved. The indicator must indicate an empty stack so that underflows are generated when the RETURN instruction transferring to the next process is executed and so the operating system can operate correctly if it makes any procedure calls.

The stacks must be emptied from the bottom for process swaps, as going through the top means interfering with the program counter at the top of its stack. Going from the bottom also is convenient because the same mechanism is used for overflow; hence, the same overflow stack can be used. The effect of process swapping will not be considered in the performance studies.

CHAPTER 3

PSP PERFORMANCE

3.1. Introduction

The parallel stack architecture uses on-chip storage to reduce the number of memory references. In this chapter, an analytical model of the architecture is presented. The model is used to compare the performance of PSP with a conventional processor. While the number of memory references is reduced, there is still considerable data movement in parameter passing which must be included in the PSP model. HLL statistics of program behavior form the basis of the model. It is shown that PSP performs considerably better than a conventional processor; execution time savings of 25% are possible. The results obtained from the model are used to determine certain design parameters of the PSP.

3.2. Performance Model

3.2.1. Overview of a Performance Model

In this section an analytical model of the PSP and of a conventional processor is described. The purpose of the model is to show quantitatively the execution time savings that are available using the parallel stack architecture. Four different situations are modeled: the PSP with parameter passing and variable allocation in registers (PSP-Reg), the PSP with parameter passing and variable allocation in memory (PSP-Mem), a conventional processor with parameter passing and variable allocation in registers (Con-Reg), and a conventional processor with parameter passing and variable allocation in memory (Con-Mem). With two strategies modeled for each processor, one can observe directly how the parallel stack architecture improves performance, independently of the variable

allocation and parameter passing strategy. Except for the absence of the stacks, Con-Reg is identical to PSP-Reg, and Con-Mem is identical to PSP-Mem.

The model is based on dynamic HLL statistics. The time, measured in CPU cycles, required for the execution of each type of HLL statement is computed and these are weighted by the frequency of that type of statement. This produces the average number of cycles required per HLL statement, which is the number used to compare the different processors and register usage strategies. Tanenbaum [Tane78] gives static and dynamic statistics for systems programs written in SAL [Tane74], a structured programming language. Other statistics are also available [PaSe82] for Pascal and C; there is good agreement between these and Tanenbaum's statistics. The SAL statistics are used because they are more complete. Dynamic rather than static measurements are used, as this indicates what is actually run on the machine.

Before going into detail of the model, the relevant characteristics of the PSP must be presented. Detailed information appears in later chapters. Both the conventional processor and the PSP use a simple instruction set with the only access to memory coming through LOAD and STORE instructions. Two cycles are required to execute a LOAD or STORE. Instruction fetch and execution are overlapped. A BRANCH instruction effectively takes one cycle as the branch is delayed until the instruction following it is executed. Instructions have three operands and execute in one processor cycle. A CALL instruction checks for overflow, pushes the stacks, and transfers to the specified address. A RETURN checks for underflow and pops the stacks using the specified mask. CALL and RETURN instructions each take two cycles. The time to process overflows is determined from the code written for that purpose. The number of registers in a processor is a parameter of the model. These registers refer to the TOSRs which are used in PSP-Reg and Con-Reg for parameter passing and variable allocation. In PSP-Mem and Con-Mem, these registers are used to hold copies of variables that have been brought in from memory. Global Registers are used only for temporary values during expression evaluation. The number of GRs is assumed large enough to

hold all temporaries that are in current use. References made to "registers" in the following description are to stack registers unless specifically stated otherwise.

3.2.2. HLL Statement Model

To compute the number of processor cycles per HLL statement, the average execution time of each statement is computed. These times are weighted by the frequency that each statement occurs. The execution time of each statement except CALL and RETURN is equal for the four situations being modeled. In other words, the only difference between the four modeled situations is in CALL and RETURN. The execution times of the other statements are needed in order to compute an overall performance measure for each modeled processor. Some changes in Tanenbaum's statistics were made to make them consistent with the analytical model. The frequency of RETURN given by Tanenbaum is the number of explicit RETURN statements. This number is much lower than the number of CALLs. Because a RETURN instruction must be executed at the end of every procedure, whether explicit or implicit, the number of CALLs must equal the number of RETURNs. For the purpose of this model, CALLs and RETURNs are treated as a pair. In order to represent this in the statistics, the frequency of RETURN was set to zero, the statistics were normalized, and the CALL frequency became the CALL-RETURN frequency. The result of this process is shown in Table 3.1 under the column labeled "Distribution."

Table 3.1. HLL Statement Distribution and Characterization.

Statement Type	Distribution	Variable References	Computation Cycles
Assignment	42.9%	2.65	1.66
IF	36.9%	2.22	3.72
CALL-RETURN	12.7%	—	—
FOR	2.2%	3.22N+2.00	4.22N+2.00
EXITLOOP	1.6%	0.00	2.00
WHILE	1.5%	2.22N	3.22N+2.00
REPEAT	0.1%	2.22N	3.22N
DO FOREVER	0.8%	0.00	1.00N
CASE	1.2%	3.00	14.00
PRINT	<0.1%	distributed among other statements	

The execution time in number of processor cycles of each statement is modeled as

(Number of Variable References)(Cycles per Variable Reference)
+ (Number of Computation Cycles).

The first term is the total time to load or store variables in registers for the statement. Each HLL statement also requires a certain number of cycles for actual computation using the registers, the second term. The formula is used for each type of statement, except CALL-RETURN pair which requires more detailed modeling. The figures in columns "Variable References" and "Computation Cycles," of Table 3.1, are computed from assembly language versions of each HLL statement and from Tanenbaum's statistics, as described below. A C compiler was used to produce VAX assembly code, which was used as a basis for producing a PSP assembly language version of each statement. The result is shown in Fig. 3.1. Each assembly language statement requires a certain number of variable references and computation cycles. The "evaluate condition" instruction in the figure is described by Tanenbaum's statement of an average of 1.22 operators per conditional expression. This results in 1.22 computation cycles and 2.22 variable references per conditional expression. For each BRANCH, an attempt is made to move an instruction after the BRANCH. This was not done when BRANCH was the first instruction for an HLL statement. Included in the computation cycles is the idle time that is present when an instruction could not be moved to after the BRANCH. In the IF statement, it is assumed that the condition is true half the time. WHILE, FOR, and REPEAT are divided into initialization and loop control. Loop control occurs every time the loop is executed; initialization occurs once per statement. In Table 3.1, the numbers for these statements are shown in two parts, with loop control multiplied by N, the average number of iterations executed per loop. As the other statistics available are from systems programs, where the average number of iterations is small, it was assumed that each loop is executed 5 times. More detailed statistics for assignment statements are available, and they are used to compute the figures for the assignment statements using a weighted average approach. Temporary space needed during statement execution is assumed to always be available in GRs. When a variable is referenced more

```
IF:                 evaluate condition
                    conditional branch to false
              true: (body)
                    branch to exit
             false: (body)

FOR:                initialize
                    branch to cond
              body: (body)
                    increment
              cond: evaluate condition
                    conditional branch to body

EXITLOOP:           branch to exit

WHILE:              branch to cond
              body: (body)
              cond: evaluate condition
                    conditional branch to body

REPEAT:       body: (body)
                    evaluate condition
                    conditional branch to body

DO FOREVER:   body: (body)
                    branch to body

CASE:               subtract lower bound from case variable
                    conditional branch to out_of_range
                    compare upper bound with case variable
                    conditional branch to out_of_range
                    load base address of table
                    add base and offset
                    load branch address from table
                    branch
                    (body)
                    branch to exit
```

Fig. 3.1. PSP assembly language versions of HLL statements.

than once in one HLL statement, all references except the first and a final WRITE are assumed to be made to registers. GRs would be used for this temporary storage. The characterization of HLL statements resulting from the above process is summarized in Table 3.1.

The number of cycles per variable reference (CPVR) is determined from operand distribution and the number of registers. Different types of operands are assumed to be found in certain places. Constant operands take as long to access as registers, as a constant is specified as a part of the instruction. Global scalars and all arrays and structures are assumed to be in memory. Accessing array and structure elements is assumed to require two variable references. Local scalars and formal parameters are assumed to be in registers if possible, otherwise in memory. Table 3.2 shows the frequencies of these operands as given by Tanenbaum. Scalars are divided into local scalars and global scalars based on 80% of scalars are local from Patterson and Séquin [PaSe82].

The time to reference local scalars and formal parameters depends on the number of registers and the number of these variables. Table 3.3 gives a summary of Tanenbaum's tables for the distribution of these variables and parameters. There is an average of 2.1 variables and 1.9 parameters per procedure. These distributions are assumed to be statistically independent. At any given time, local scalars and formal parameters are in registers, except when there are not enough registers. The fraction of local scalars and formal parameters found in stack registers (FLR) in a given procedure is

Table 3.2. Distribution of Operand Types.

Constant	32.8%
Local Scalar	33.5%
Array/Structure	20.3%
Global Scalar	8.4%
Bit Field	3.3%
Function Call	1.6%

Table 3.3. Distribution of Number of Local Scalar Variables and Number of Formal Parameters per Procedure.

N	Percent of Procedures with	
	N Local Scalars	N Formal Parameters
0-1	57.4%	48.7%
2-3	19.6%	34.0%
4-5	14.9%	15.4%
>5	8.1%	1.8%

$$FLR = \begin{cases} \min(1, R/l) & \text{if } l \neq 0 \\ 0 & \text{if } l = 0 \end{cases}.$$

The number of TOSRs, R, does not include two reserved registers for FP and a return value. The number of variables local to a procedure, l, includes locally defined scalar variables and parameters that were passed to the procedure. Assuming a uniform distribution of access to these variables the average time to reference one is

$$CPVR_{one} = FLR \cdot RAT + (1-FLR) \cdot MAT$$

in a given procedure. The register access time, RAT, is equal to zero as the actual use of the variable is taken into account as computation cycles. The memory access time, MAT, is the time to load or store a register. The average time to reference local scalars is

$$CPVR_{local\ scalar} = \frac{\sum_l \text{freq}(l) \cdot l \cdot CPVR_{one}}{\sum_l \text{freq}(l) \cdot l}.$$

This is the total time for referencing divided by the number of references. The frequency of procedures with l local scalars and formal parameters is freq(l). Finally, the overall cycles per variable reference is computed by weighting the time to reference each type of variable by the frequency of the variable (Table 3.2).

The cycles per variable reference is the same for all four cases that are modeled. When variables are allocated in registers and parameters are passed in memory, registers must be initialized in order to achieve the cycles per variable reference. This initialization is included in the CALL. No initialization is required for the register passing and allocation strategy. The model does not include register optimization which would put the most frequently accessed variables in registers for the period of frequent access. Once registers are assigned at the beginning of a procedure, the register assignment is fixed.

3.2.3. Procedure Call and Return Model

The above description of the model is identical for all four cases being modeled. CALL and RETURN execution times are combined with other statement execution times in the formula given by setting the number of variable references in a CALL and RETURN to zero. The number of computation cycles includes everything needed for CALL and RETURN. There are many terms added to arrive at an average execution time of a CALL and RETURN for each processor and register usage strategy. The terms are: basic CALL and RETURN, overflow and underflow, parameter passing, register saving and restoring, register permuting, *call-by-reference* parameter restoring, and initialization. Each of these is described below.

In some of the terms that follow, especially for PSP-Reg, it is necessary to know more detailed information about which registers contain variables that will be passed as parameters and which registers contain other variables. It is assumed that parameters to be passed could come from any of the current formal parameters or local scalars with equal probability and from any register used by a procedure with equal probability. This probability information can be used with the knowledge of how registers are assigned at the beginning of a procedure. It is assumed first, that two registers are reserved, in all four cases being modeled, for the FP and a return value. Next, for PSP-Reg, registers are reserved for parameters and finally for local variables. Specifically, registers are assigned in sequential order for parameters then for local variables.

For each modeled situation the average time to execute a CALL-RETURN is computed. For a given number of registers, R, the average time to execute a CALL-RETURN is computed by finding the average time for each combination of l, the number of local variables and formal parameters, and p, the number of actual parameters to be passed, and weighting these by the frequency, $\text{freq}(l) \cdot \text{freq}(p)$.

3.2.3.1. Parameter Independent Terms

The basic CALL includes the time to compute the CALL address and to transfer to the new procedure, which is one CALL instruction. A new FP must be computed and the old one saved. The old PC and FP are saved in register stacks for the PSP and are saved in memory for the conventional processor. The basic RETURN is the time to restore the PC and FP and return to the calling procedure.

Overflow and underflow terms are relevant only for the PSP. The number of cycles required to handle overflow and underflow is determined from the code of the actual routines. The number of cycles is taken from the worst case time to execute these routines (see section 5.3). For the underflow routine this includes checking every bit of the mask to determine what TOSRs need to be loaded. The execution times of the overflow and underflow routines are multiplied by the frequency of overflow and added to the basic CALL and RETURN. This overflow frequency is directly related to the depth of the register stacks.

3.2.3.2. Parameter Passing

The time to pass parameters is divided into two parts: parameter fetching, moving the data to a processor register, and moving the parameter close to its final location. Since all memory-to-memory operations must go through a processor register, this is a natural division. The time to fetch parameters is computed exactly as the cycles per variable reference is computed; a distribution of operand types and the location of each operand is used to compute the time to reference a parameter. A difference from an ordinary variable reference is that for some parameters only the address is needed. The time to fetch parameters reflects the different methods of passing *call-by-reference* parameters. The distribution of the type of variables used for parameter passing is shown in Table 3.4. The labels in parentheses in the table are used to represent the fraction of parameters that are of the indicated type. The values in the table are educated guesses from general program behavior since the desired statistics are not available. The fraction of parameters that

Table 3.4. Distribution of Parameter Types in a CALL.

Local Scalar (BLS)		55%
call-by-value	33%	
call-by-reference	22%	
Temporary (BT)		20%
Array/Structure Name (BAN)		10%
Array/Structure Element (BAE)		5%
call-by-value	4%	
call-by-reference	1%	
Global Scalar (BGS)		5%
Constant (BC)		5%

can be found in registers (BR) is

$$BR = BLS \cdot FLR.$$

The fraction of scalar parameters in memory (BM) is

$$BM = BLS \cdot (1-FLR) + BGS.$$

All other parameters are found in memory.

Moving a parameter close to its proper location is moving the parameter to its proper memory location or moving it to any TOSR if it is to be passed in a register. The time for parameter fetching and movement is the same for PSP-Reg and Con-Reg, and the same for PSP-Mem and Con-Mem. The fraction of parameters that are passed in registers (PR) is

$$PR = \begin{cases} \min(1, R/p) & \text{if } p \neq 0 \\ 0 & \text{if } p = 0 \end{cases}$$

where p is the number of parameters to be passed. For PSP-Mem and Con-Mem, however, $PR = 0$. The fraction in memory (PM) is $1-PR$. The fraction of parameters that are *call-by-reference* (PREF) and *call-by-value* (PV) are functions of the type of the parameter. The value of PV is 1 for all types except scalars and array elements as indicated in Table 3.4.

Each parameter is characterized by its type, where it is passed, and whether the value can be changed. The number of parameters of a specific characterization is the product of the fractions of each of the three characteristics and the total number of parameters in a particular CALL. For

example, the number of constant parameters passed by value in registers is $BC \cdot PR \cdot PV(constant) \cdot p$. Each parameter requires a certain number of cycles for parameter passing. Table 3.5 shows the time required for each type of parameter in PSP-Reg. The time is presented as a description of what must be done for each parameter. The table includes parameter passing as well as other terms to be described later. Table 3.6 shows the case for PSP-Mem. Con-Reg and Con-Mem are similar but do not use stacks for saving registers. PSP-Reg and Con-Reg, when there are not enough registers for parameter passing, use memory for the extra parameters in

Table 3.5. Parameter Passing in Registers.

Passed in	Type of Parameter (Frequency of Parameter Type)					
	Constant	Local (BLS) and Global (BGS) Scalars		Array Name	Array Element	Temporary
	(BC)	Register (BR)	Memory (BM)	(BAN)	(BAE)	(BT)
Register by value (PR×PV(type))	CALL: 1 RETURN: 0	permute permute	BVI 0	1 0	1+BVI 0	1 0
Register by reference (PR×PREF(type))	—	permute permute	BVI BV	—	1+BVI 1+BV	—
Memory by value (PM×PV(type))	1+AV 0	AV+CV CVI	BVI+AV 0	1+AA 0	1+BVI+AV 0	AV 0
Memory by reference (PM×PREF(type))	—	BV+AA BVI	1+AA 0	—	2+AA 0	—

Key:
First letter:
A: Register to memory move, as a parameter
B: Register to other memory move
C: Save register contents in another register if possible, else in memory
Second letter:
A: address
V: value
Optional Third letter:
I: inverse operation of first letter specification

Table 3.6. Parameter Passing in Memory.

Passed in	Type of Parameter (Frequency of Parameter Type)					
	Constant (BC)	Local (BLS) and Global (BGS) Scalars		Array Name (BAN)	Array Element (BAE)	Temporary (BT)
		Register (BR)	Memory (BM)			
Memory by value (PV(type))	CALL: 1+AV RETURN: 0	AV 0	AVI+AV 0	1+AA 0	1+AVI+AV 0	AV 0
Memory by reference (PREF(type))	—	1+AA 0	1+AA 0	—	2+AA 0	—
Key:	see Table 3.5					

the same way that PSP-Mem and Con-Mem use memory for parameter passing.

3.2.3.3. Register Saving, Restoring, and Permuting

There is register saving on a CALL on restoring on a RETURN for all four cases studied. Computation of this is different for different cases. For PSP-Mem, all saving is included in the basic CALL during the stack push. For Con-Mem, all registers used by the called procedure must be saved in memory. However, as some register optimization may reduce the number of registers needed, only 75% of the local scalars and formal parameters are saved. The same registers that were saved on a CALL are restored on a RETURN.

Register saving and permutation in PSP-Reg and Con-Reg are part of register saving and parameter passing. Permutation involves moving parameters to the correct registers from other registers. At the point the CALL is issued, all parameters must be in the proper locations. In PSP-Reg, registers not used to pass parameters may contain other variables which are saved by pushing the stacks. In Con-Reg, these other variables must be saved in memory. Register saving in PSP-Reg is used to make room for parameters in the parameter registers. A parameter register is a register that is to be used to pass a parameter in the current CALL. In PSP-Reg, register saving includes

some register-to-register and register-to-memory movement in order to free some parameter registers. When there are parameters in non-parameter registers, some parameter registers do not have to be freed until the register permutation phase. Some additional register saving is needed for the way parameters are passed. For example, for a local variable in a register that is to be a *call-by-value* parameter in memory, a copy must be kept for use after the RETURN (see Table 3.5). Following the RETURN, the registers are restored and re-permuted to the state before the calling sequence.

In order to compute the time for register saving and permutation, more knowledge of the distribution of variables in registers is needed. The contents of a register can be an actual parameter, some other local variable, or empty. Empty registers are those not currently in use and are grouped together at one end of the register set due to the register assignment algorithm used. An actual parameter coming from a TOSR is assumed to come from any TOSR currently in use with equal probability. The fraction of actual parameters that are found in parameter registers, given that the parameter is in a register (FPPR), is

$$FPPR = \begin{cases} \min(1, PReg/LReg) & \text{if } LReg \neq 0 \\ 0 & \text{if } LReg = 0 \end{cases}$$

where $PReg$ is the number of parameter registers ($\min(R, p)$) and $LReg$ is the number of local scalars in registers ($\min(R, l)$). The number of parameters initially in parameter registers and to be passed in parameter registers (PPR) is

$$PPR = FPPR \cdot BR \cdot PR \cdot p.$$

The total number of parameter registers that must be vacated is $PReg - PPR$.

The number of free registers is initially

$$Free = R - \max(LReg, PReg).$$

There may be parameters in non-parameter registers. If the parameter is to be passed in memory, this register will become free. The number of free registers is increased to

$$Free' = Free + (1-FPPR) \cdot BR \cdot PM \cdot p.$$

Register saving in PSP-Reg moves all non-parameter variables in parameter registers to free registers if there is room. Other non-parameter variables are moved to memory unless there are non-parameter registers containing parameters. These non-parameter registers will be available to save non-parameter variables. Moving these variables and parameters is included in register permutation.

After register saving and memory parameter passing have been completed, the contents of the parameter registers are as follows: there are parameters which are not necessarily in the correct registers; there are other variables which still must be removed; and there may be empty registers due to saving registers or passing parameters in memory. In non-parameter registers there may be some parameters yet to be moved to parameter registers and there may be other variables. Register permutation moves register contents to their correct locations. Based on the register saving and parameter passing, the number of registers in each of the above categories can be determined from probability. The permutation time is computed by knowing the number of registers to take part in the permutation, and the number of these registers that are empty. The average time for each case, based on the assumption that any combination of registers may be empty with equal likelihood, was computed and used in the model. Fig. 3.2 gives an example calculation of one particular case. Other cases are computed similarly.

More information is known, however. Finding the average time assumes that all possibilities are equally likely. There are some that cannot occur. For example, two parameters would not be interchanged if one or both are in non-parameter registers. Each parameter in a non-parameter register requires one cycle to move to a parameter register. Similarly, each non-parameter is moved to a non-parameter register in one cycle. If there are no free parameter registers, one must be created to achieve the above results, requiring one register-to-register move to free a register and one to restore it. While this permutation is taking place, permutation of parameters in parameter registers occurs making use of free registers when they occur. This part of the permutation is

Description		Probability	Time
Correct	A B	0.167	0 cycles
Interchanged	B A	0.167	3
One Correct	A / B A	0.333	1
Neither Correct	A B / B A	0.333	2
	Average of 1.5 cycles		

Fig. 3.2. An example permutation calculation of two variables in three registers.

estimated using *PReg* registers of which *PPR* are occupied using the process illustrated in Fig. 3.2.

The above calling sequence for PSP-Reg is designed to reduce permutation. It does this by first emptying as many registers as possible due to parameter passing in memory and register saving. The remaining registers are permuted. Any parameter or address that must be loaded into a register is then done. For Con-Reg, a similar calling sequence is used. The major difference is that registers cannot be used to save variables. All non-parameter variables in registers are moved to memory. Parameters are then moved to the proper registers. More time is spent moving variables, but less time is required for permutation as there are fewer variables that take part.

3.2.3.4. Other Terms

In PSP-Reg and Con-Reg, some extra cycles may be needed on RETURN due to the method of parameter passing. When a variable is allocated in memory but is a *call-by-reference* parameter in a register, the changed value must be moved back to memory after the RETURN.

PSP-Mem and Con-Mem require some initialization upon entry to the procedure. This is included in the CALL. This initialization is needed to achieve the cycles per variable reference used when referencing variables. Local scalars do not need to be initialized, as they have no value at the beginning of the procedure; local scalars that are to be referenced from registers can be thought to have already been moved to a register, assuming the first reference is a WRITE. Formal parameters that are to be referenced from registers must first be loaded from memory. The number of these parameters that need to be moved is computed by assuming that the probability of a parameter being moved to a register for initialization is the same as the ratio of the number of local scalars and parameters in registers to the total number of local scalars and parameters ($LReg/l$).

3.3. Results from the Performance Model

The model was used to determine the performance of both processors, each with both parameter passing and allocation strategies. The number of TOSRs was varied to see how this changes the performance. The number of TOSRs is the abscissa of the performance graphs. For a conventional processor this is the number of registers that are available for parameter passing and variable allocation or for use as copies of variables allocated in memory. For both processors, it must be remembered that additional registers are assumed available for use during expression evaluation. The number of registers in the following graphs begins at two because two registers are always reserved for the FP and a possible return value. Performance is given in CPU cycles, the ordinate in the graphs. Better performance is fewer processor cycles. An overflow percentage is the percent of CALLs which cause an overflow.

Fig. 3.3 shows the average number of cycles to reference a variable. This is the same for each of the four modeled situations. The curve shows, as expected, that it is better to have more registers during procedure execution. The magnitude of the slope of the curve decreases with more registers. This is because there frequently is not a need for more registers.

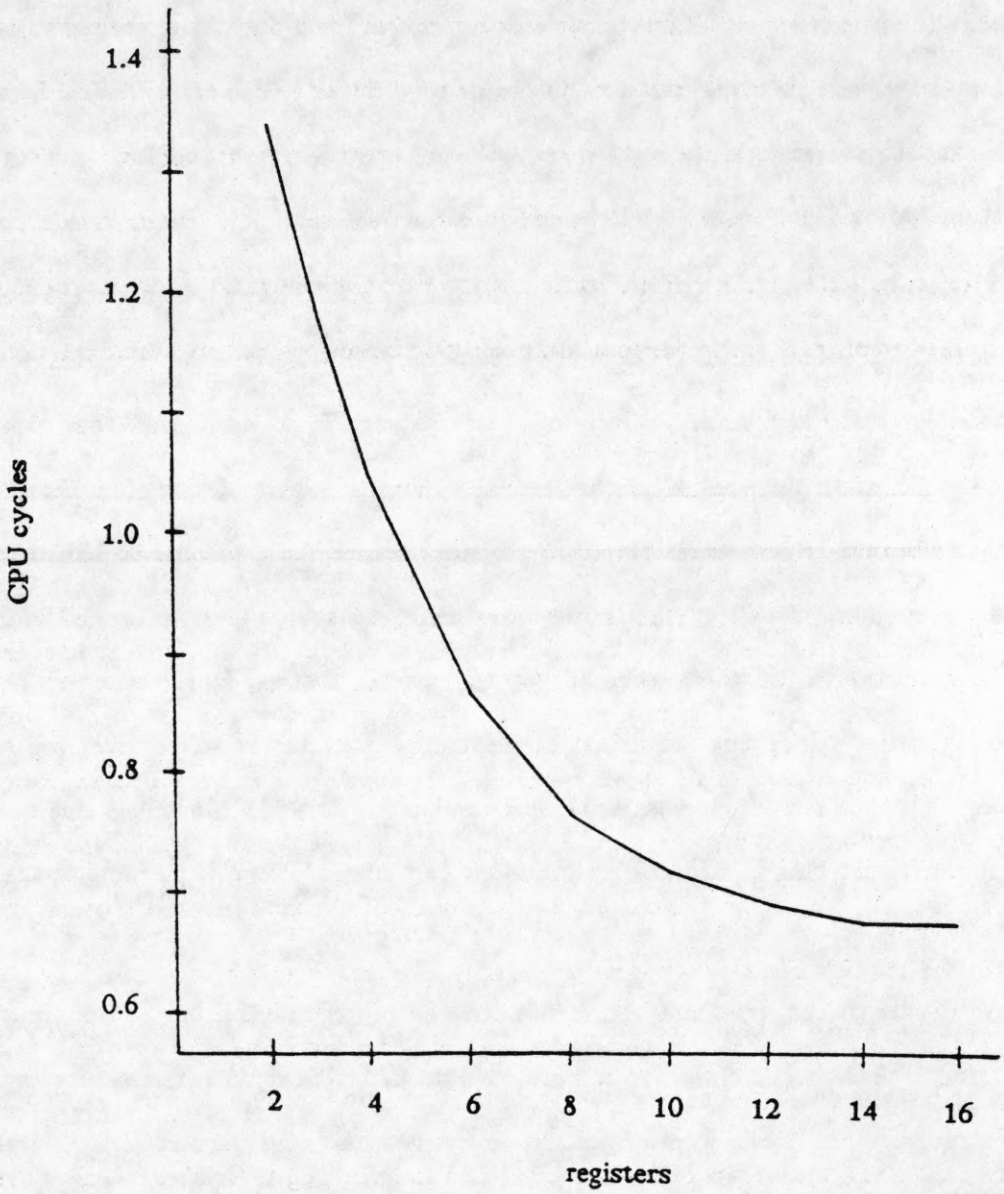

Fig. 3.3. Relationship between number of registers and average time to reference a variable. Two-cycle memory access.

The time to execute a procedure call is shown in Fig. 3.4. PSP-Reg is the only curve that decreases with more registers. The others increase because there are more registers to save or initialize. While there is some data movement needed for PSP-Reg, most of that is from register to register, while the other situations require many moves from register to memory. Because PSP-Reg and Con-Reg revert to using memory in the same way as PSP-Mem and Con-Mem for variables that do not fit in registers, the curves for PSP-Reg and PSP-Mem and for Con-Reg and Con-Mem meet at one point. This occurs in all the graphs presented here.

A similar graph for RETURN is shown in Fig. 3.5. The most important observation to be made is that PSP-Mem and Con-Mem are faster than PSP-Reg and Con-Reg, respectively. When a parameter is passed in memory, a copy of the parameter or the address of the parameter is passed. There is no need to use this after the RETURN. In PSP-Reg and Con-Reg, because many of the parameters would have been allocated in registers before the CALL, they had to be moved from register to register. Much of this same movement is also required on a RETURN. Because the underflow routine must check each bit of the mask, it takes more time per register than the overflow routine. Underflow becomes the most important factor with increasing registers for the PSP-Reg. All four curves increase with more registers. Con-Mem increases due to restoring registers. Register restoring is also the predominant part of Con-Reg. Underflow is the only factor in the PSP-Mem curve.

As discussed earlier, there are a number of different terms used to compute the time for CALL and RETURN. Table 3.7 summarizes what must be done for each processor and register usage strategy. The table gives an indication of how the average number of cycles required for each factor is affected by the number of registers. Register saving and restoring times increase in Con-Reg and Con-Mem because more registers are used when they are available and these must be stored in memory. PSP-Mem saves its registers on the parallel stacks, requiring no additional time. PSP-Reg saves some registers in other empty registers. With few registers, all registers are likely to

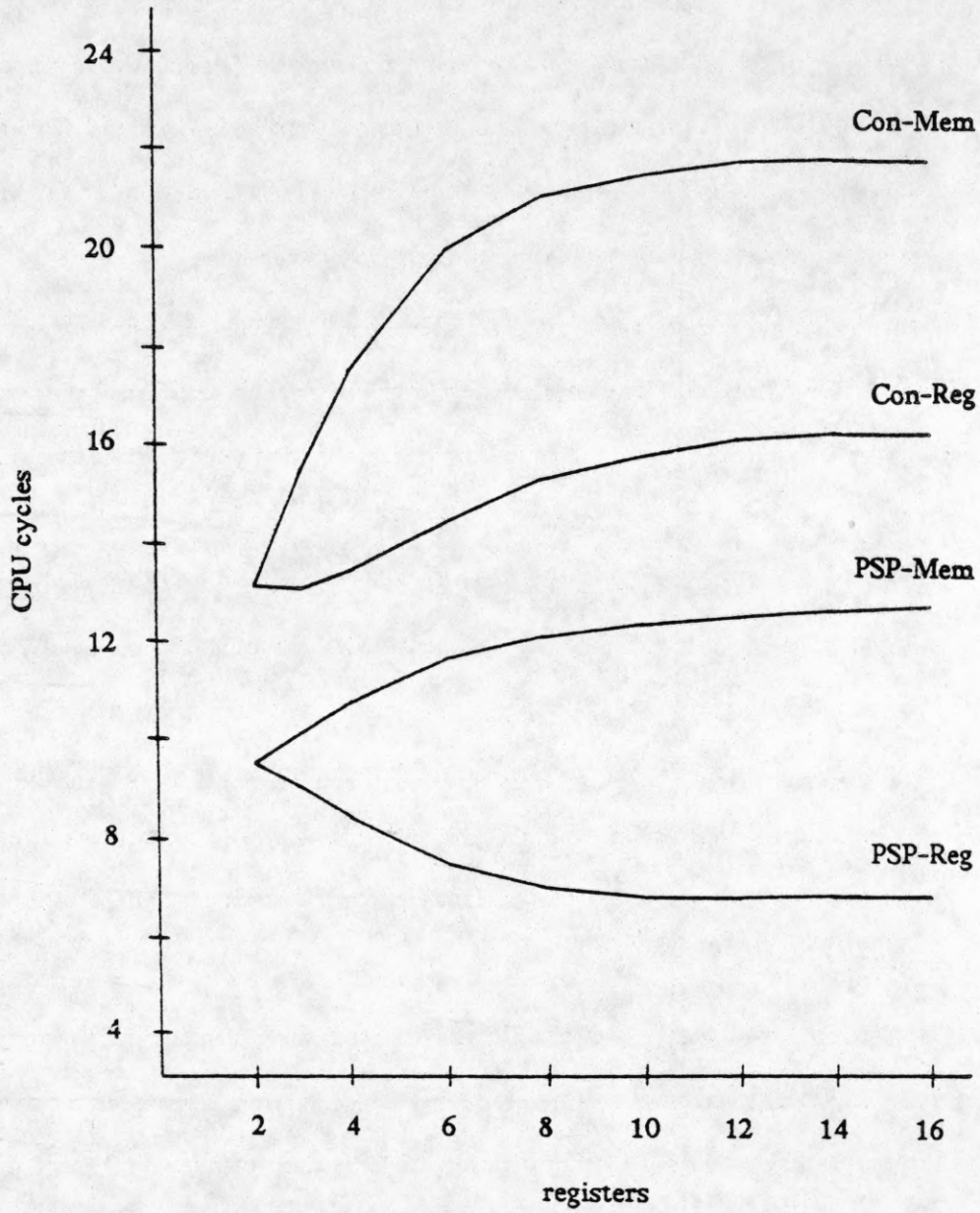

Fig. 3.4. Relationship between number of registers and average time to execute a CALL. Two-cycle memory access. Two percent overflow for PSP.

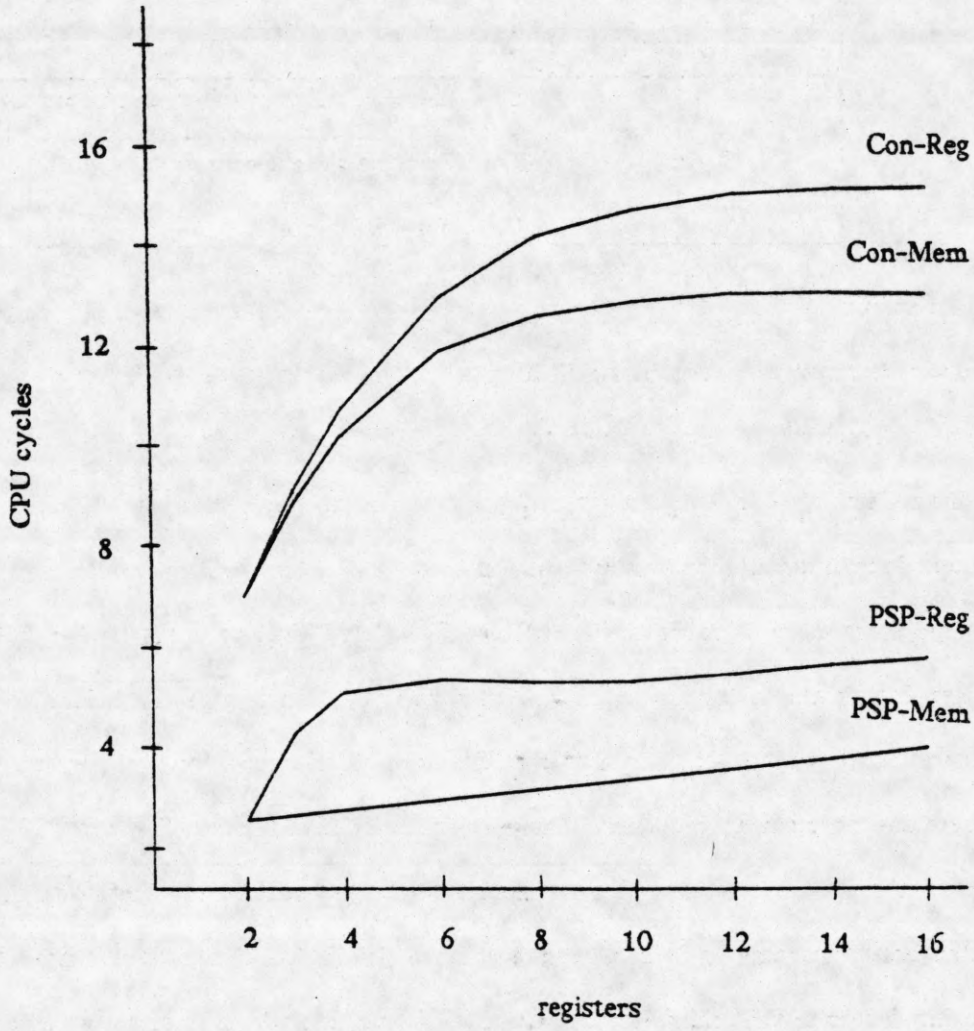

Fig. 3.5. Relationship between number of registers and average time to execute a RETURN. Two-cycle memory access. Two percent underflow for PSP.

Table 3.7. Effect of an Increasing Number of Registers on CALL and RETURN Model Components.

Component	Effect of Increasing Number of Registers							
	PSP-Reg		PSP-Mem		Con-Reg		Con-Mem	
	C	R	C	R	C	R	C	R
Save Registers	ID	0	0	0	I	0	I	0
Restore Registers	0	ID	0	0	0	I	0	I
Fetch Parameters	D	0	D	0	D	0	D	0
Move Parameters	D	0	N	0	D	0	N	0
Permute Registers	ID	ID	0	0	I	I	0	0
Restore Call-by-Ref	0	ID	0	0	0	ID	0	0
Initialize Registers	0	0	I	0	0	0	I	0
Handle Overflow	I	0	I	0	0	0	0	0
Handle Underflow	0	I	0	I	0	0	0	0

Key:		
	D	decrease number of processor cycles
	I	increase number of processor cycles
	ID	increase then decrease
	N	no change
	0	not applicable

be used so it takes longer to save registers as more registers are added. There comes a point, however, where more registers result in a better chance that more are empty so the time decreases. Parameter fetching is very similar to cycles per variable reference which decreases. Parameter moving reflects how many parameters go to registers and how many to memory. As the number of registers increases, more parameters are passed in registers and more come from registers. With more registers there is more to permute in Con-Reg. The same is true for PSP-Reg, but there is another factor added—some other variables take part in the permutation. When there are few registers there is a greater chance of interference. With more registers permutation time decreases as there are more empty registers. The permutation times for Con-Reg and PSP-Reg converge as the number of registers increases. When restoring *call-by-reference* parameters there is a greater chance that an actual parameter is in a register before the CALL, and will not need to be restored. An opposing factor is that more actual parameters that start in memory are passed in registers. The decrease with increasing registers reflects the interaction of these two factors. Initialization and overflow and underflow handling increase with more registers.

The cycles per variable reference, cycles per CALL, and cycles per RETURN can be combined to produce an overall measure of performance. This measure, the average processing time per HLL statement, is shown in Fig. 3.6. The curve for PSP-Reg decreases rapidly as more registers are added. PSP-Mem also shows a decrease, although not as great. The increasing number of cycles for a CALL is the cause of this. Both curves begin to increase, as the increase in the overflow penalty becomes greater than the decrease in the other factors. The minimum point for PSP-Reg is at 14 registers and for PSP-Mem at 12 registers. The Con-Reg and Con-Mem show little decrease—in fact there are some increasing sections of both curves. The rapid increase in the time for a CALL and RETURN balances the decrease in the cycles per variable reference curve. This demonstrates the trade-off of using registers to decrease reference time, but having to save these registers on a procedure call. It should be pointed out that in a conventional machine, the entire register set would be treated the same. In this model, there are two distinct types of registers—temporary data registers and variable and parameters registers. If all registers were treated equally, one would expect monotonic curves for the two cases. Whether they are increasing or decreasing depends on how the number of variables referenced is related to the number of procedure calls. With 8 registers, there is a 25% performance improvement using PSP-Reg over Con-Reg and using PSP-Mem over Con-Mem. PSP-Reg is 5% better than PSP-Mem at that point.

Fig. 3.7 shows how the average time to execute a statement for PSP-Reg changes as the percent of CALLs that result in overflow changes. Since the overflow is a function of stack depth, Fig. 3.7 indirectly shows the effect of stack depth on performance. One can see that for 0% overflow, the curve continues to decrease, while the others reach a minimum at 14 and 12 registers. In practice one cannot achieve 0% overflow so 16 registers are not needed. With 2% overflow, the performance with 8 registers is within 3% of the best performance but only requires two-thirds the number of registers. A stack depth of 8 can achieve a 2% stack overflow [HaKe80]. The PSP design, therefore, uses 8 parallel register stacks each of depth 8. When too many registers are included, there could be a significant waste of these registers as few procedures need so many registers. There is waste

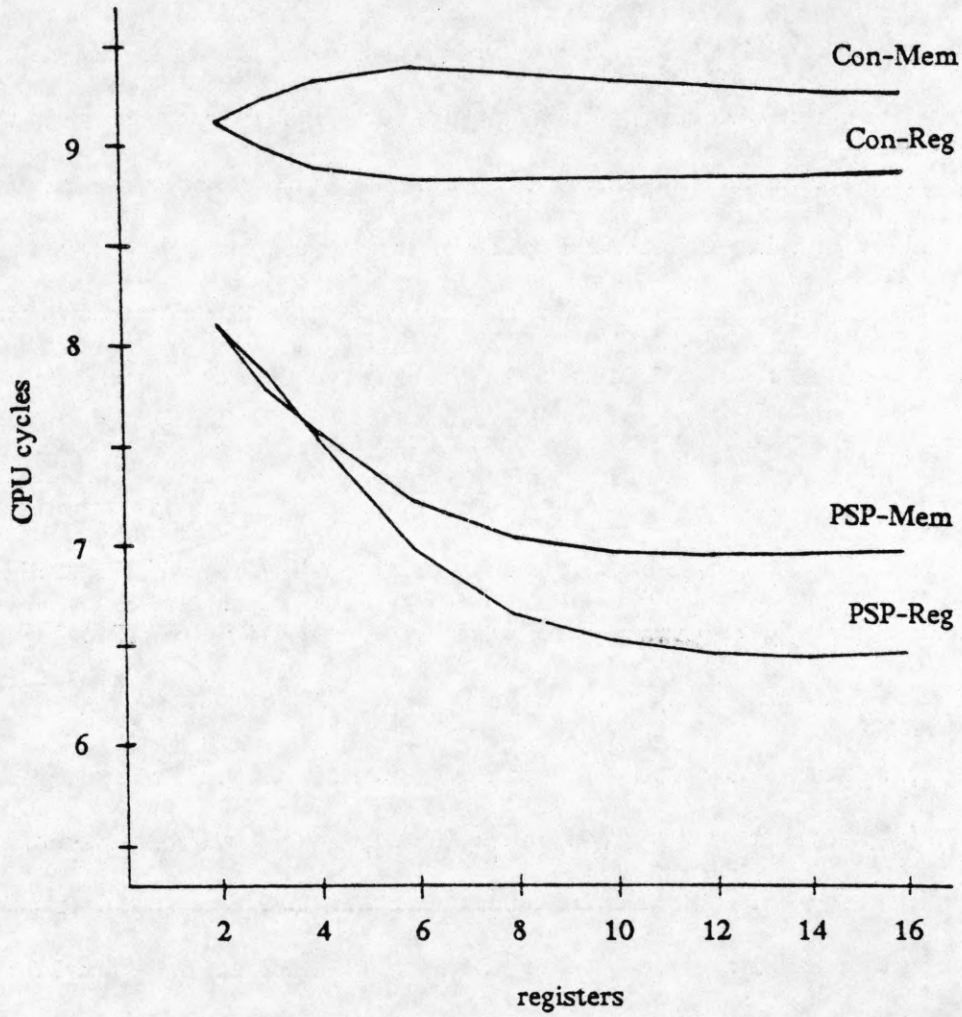

Fig. 3.6. Relationship between number of registers and average time to execute a HLL statement. Two-cycle memory access. Two percent overflow for PSP.

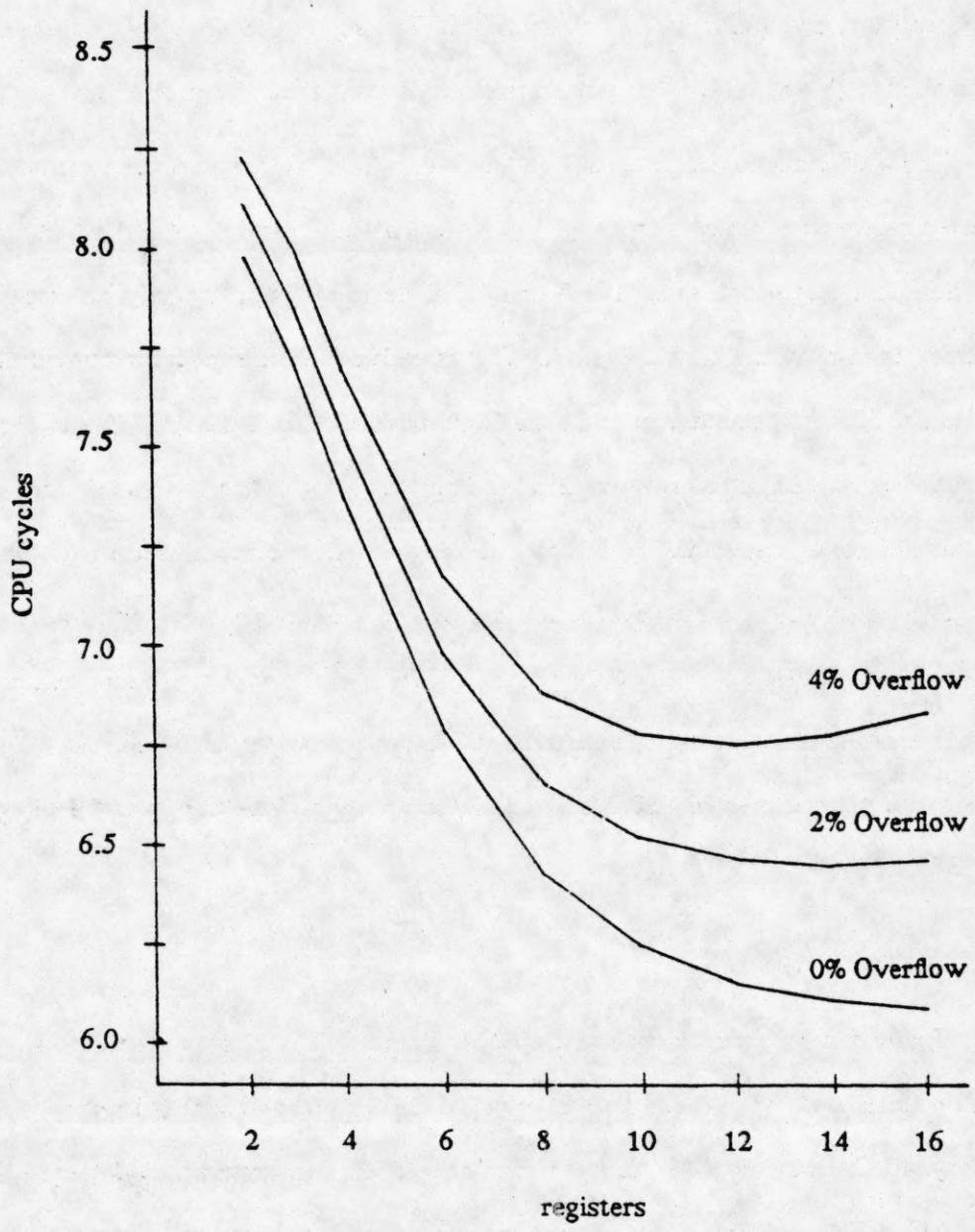

Fig. 3.7. Effect of overflow on average time to execute a HLL statement for PSP-Reg. Two-cycle memory access.

because the amount of on-chip storage reserved for each procedure is fixed. This waste affects the CALL overflow and RETURN underflow performance because it results in unnecessary data movement of unused registers. Two registers are used for FP and return value and there is an average of two locals and two parameters per procedure. The eight registers include these six plus two others. With eight registers, 82% of all procedures can fit their parameters and local scalars in registers.

The performance described so far has been with a fairly fast memory speed compared to the register referencing time. Figure 3.8 shows the average time to execute a statement when LOAD and STORE instructions each take four cycles. In the model, the only other penalty of the slower memory is in branches that are taken. This is included in the model in the number of computation cycles for HLL statements, in the time for a basic CALL and RETURN, and in the overflow and underflow routines. The processor modeled for the slow memory was assumed able to execute one instruction per cycle, either by fetching more than one instruction at a time or by using an interleaved memory to fetch one instruction per cycle. The curves are very similar in shape to the faster memory of Fig. 3.6. The only major differences are that the distance between the curves is greater and all curves have been moved up—have worse performance. At 8 registers, PSP-Reg is 29% better than Con-Reg, PSP-Mem is 28% better than Con-Mem, and PSP-Reg is 9% better than PSP-Mem.

3.4. Observations

Changes were made in the statistics used in the model to see how sensitive the model is to these changes. Only two changes in input made significant changes in output. Reducing the frequency of CALLs improves the performance of all four cases and decreases the space between them. Con-Reg and Con-Mem curves show a definite decrease with more registers. These effects occur because the ratio of the number of variable references to the number of CALLs and RETURNs increases. The time to execute a statement becomes more like the cycles per variable reference curve. For example, at 8 registers and 2% overflow, with the CALL frequency cut in half

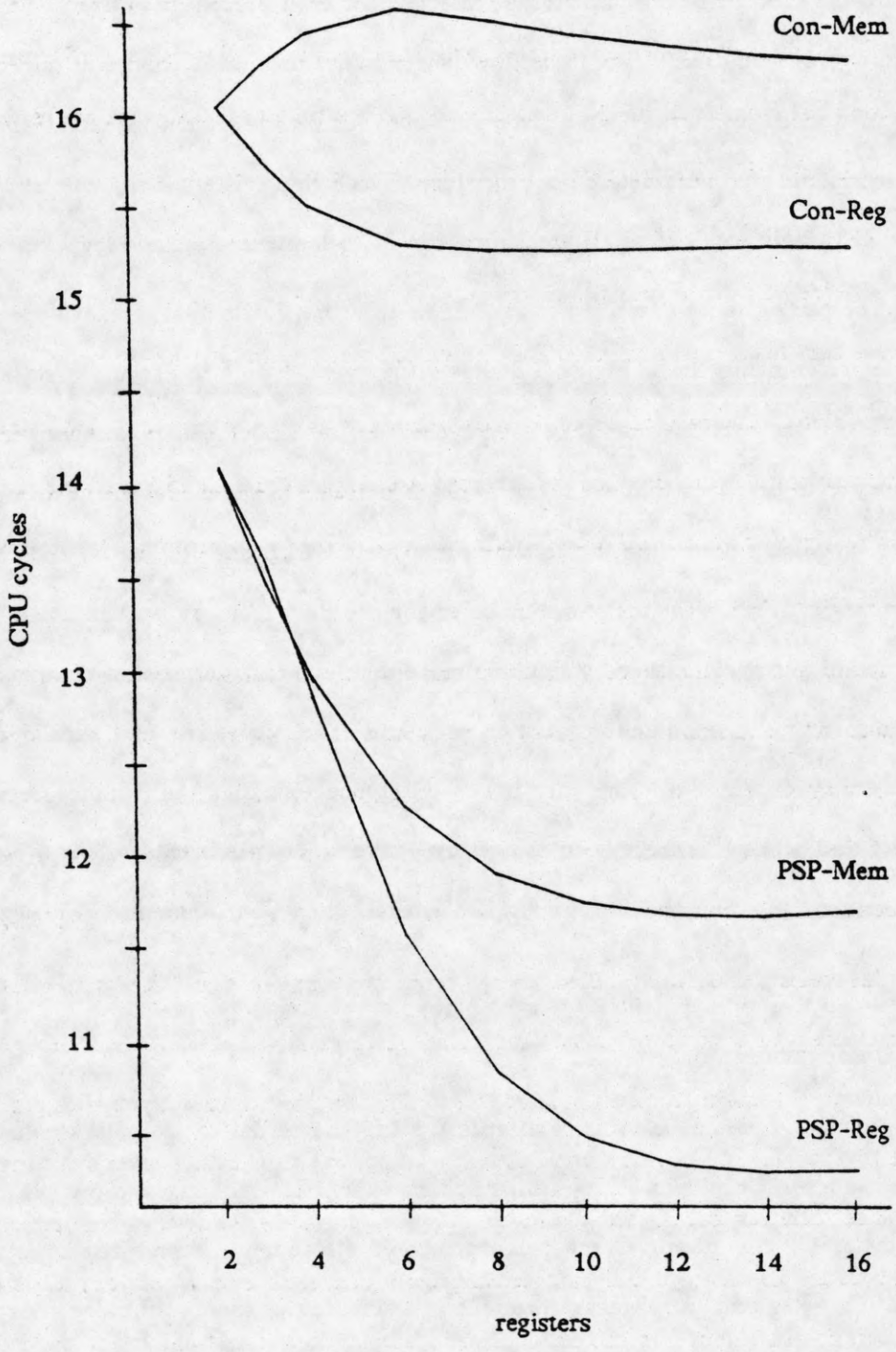

Fig. 3.8. Relationship between number of registers and average time to execute a procedure call. Four-cycle memory access. Two percent overflow for PSP.

(with subsequent increases in other statement frequencies), PSP-Reg performance increases by 6%. Similarly, Con-Reg performance increases by 19%, but PSP-Reg is still 15% better than Con-Reg. Increasing the CALL-RETURN frequency causes the time per statement curve to become more like the CALL and RETURN curves, which increase. For PSP-Reg and PSP-Mem the same effect can be seen. The difference is that for an increase in the CALL-RETURN frequency, PSP-Reg would still show a decrease with increasing registers. With the CALL frequency doubled, the performance of all processors decreases, but PSP-Reg is now 36% better than Con-Reg.

A second, but related result is seen when the average number of loop iterations is changed. An increase causes performance to decrease. A change from 5 to 15 iterations causes a 35% decrease in PSP-Reg performance and a 26% decrease in Con-Reg performance. PSP-Reg performance is 20% better than Con-Reg, however.

There are a number of factors which are not included in the model. Interrupts are not included. These are similar to CALLs and would affect a conventional processor much more than PSP. A context switch would have the opposite effect. Efficient use of moving statements after BRANCHes in the HLL statement model is not made as there is no interaction between the assembly code from different statements. This would reduce the execution time equally for all machines.

The variable allocation algorithm assigns registers once and keeps the assignment fixed. A better strategy is to keep the most frequently referenced variables in registers over the time period in which they remain frequently referenced. This would improve the cycles per variable reference of all four cases equally, however. A similar gain could be achieved in the time to fetch parameters. The time for parameter passing in registers could be reduced if register allocation were not fixed, because allocation could include choosing which register a variable is to be allocated in, based on where it should be for the next procedure call. Registers would not have to be re-permuted after a RETURN in most cases. A better assumption for register usage in PSP-Reg and Con-Reg would be to allocate parameters from one end of the register set and variables from the other end.

This would put free registers where they may more likely be used for subsequent parameter passing. Currently the free registers are at one end of the register set. There was one register reserved for a return value in all procedure calls. If this register could be used for variable allocation in procedures that did not return a value, it would be equivalent to having one more register. This would help PSP more as it shows better performance with more registers.

The time to indirectly reference *call-by-reference* parameters passed as an address in memory is not included. This would slow PSP-Mem and Con-Mem. There is no attempt to take into account referencing variables declared in intermediate level procedures. This requires allocating the variable in memory for PSP. While this would slow PSP more than a conventional processor, it is infrequent [Weic84]. The worst-case time was used for a RETURN underflow. Both PSP-Reg and PSP-Mem would perform slightly better if a more realistic time were used. Because PSP-Mem does not use the mask, except for a return value, the RETURN underflow routine would not have to check every bit of the mask. This would improve PSP-Mem performance somewhat.

The cycle time of a conventional processor, as described in here, could be less than that of PSP. This would have a direct effect on overall performance. PSP may have long busses, due to the large register file, which may require a longer time to change state due to capacitive effects.

The above factors would have an effect on performance. The amount cannot be determined without more statistics and a more complicated model. However, more of these factors favor PSP over a conventional processor, so one might expect the relative PSP performance to be even better than the model indicates.

CHAPTER 4

PROCESSOR ARCHITECTURE AND ORGANIZATION

4.1. Introduction

The parallel stack architecture was described in the preceding chapters. In this chapter, the architecture and organization of a processor that includes the parallel stacks will be discussed. The Parallel Stack Processor is a register machine with 32-bit data paths. Only LOAD and STORE instructions access memory. Instruction fetch and execution are overlapped. The capabilities of the data path are considered here, while the actual instruction set is presented in the next chapter.

4.2. Data Path Description

The data path organization is given in Fig. 4.1. There are eight stack registers, numbered 8 through 15. In addition, eight more registers, the Global Registers, numbered 0 through 7, are provided without stacks. Each of the 16 registers has two READ ports and one WRITE port. Registers are read to busses A and B, which take operands to be processed, and written from bus C, which has the result of an operation. The process of read, execute, and write is defined as one instruction cycle. A single instruction specifies the two source registers, the operation, and the destination register. Three different registers may be used. The source data do not have to come from a TOSR or GR, nor are the destination data necessarily written to one of these registers. The coding of operands in the instruction allows other parts of the processor to be accessed.

Arithmetic, logical, and shift operations can take place in the execution phase of an instruction cycle. Two function units are provided. An Arithmetic and Logical Unit (ALU) does addition, subtraction, AND, OR, and EXCLUSIVE-OR. Both signed and unsigned arithmetic can be

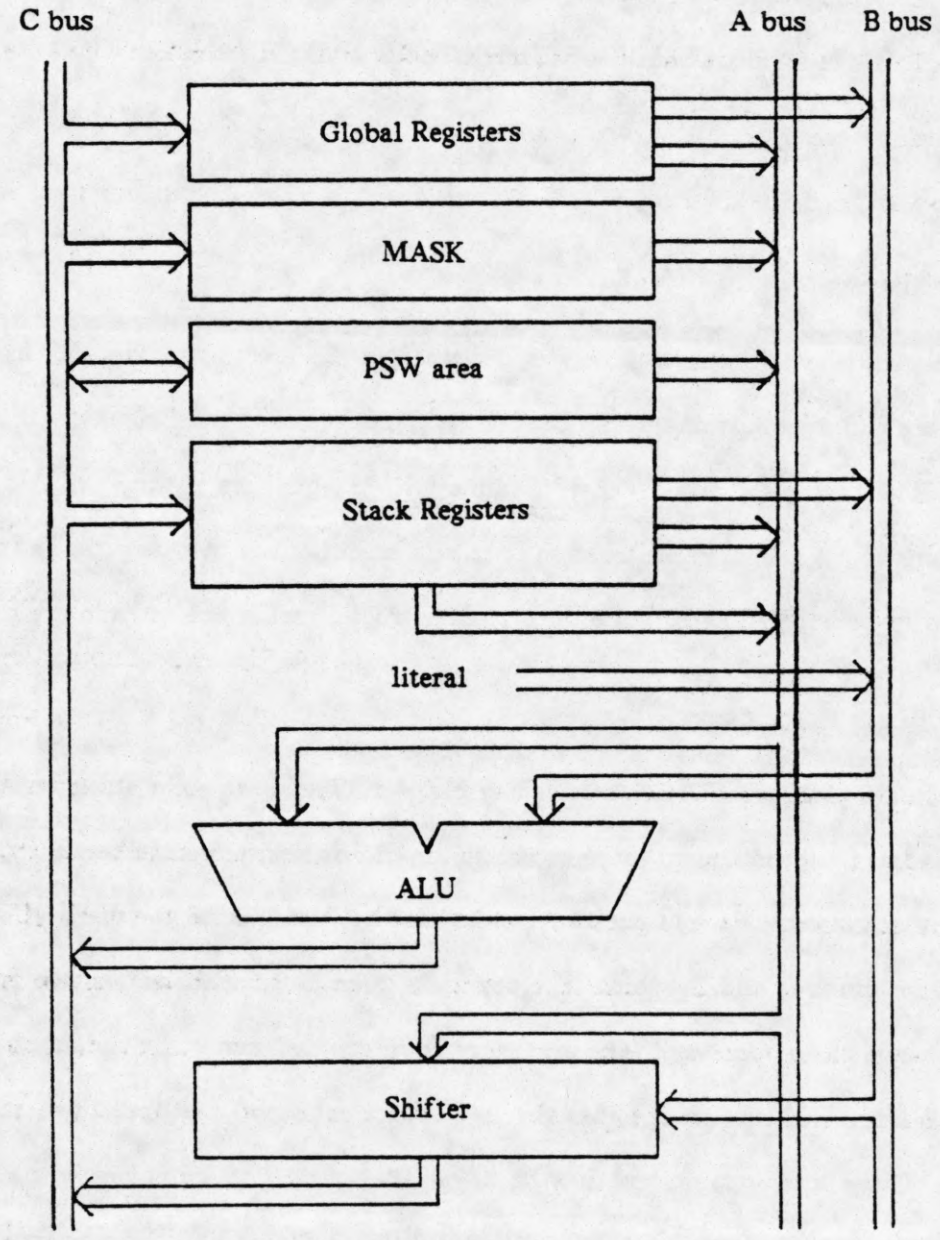

Fig. 4.1. CPU organization.

done. A shifter can do left and right, arithmetic and logical, shifts of A bus operand of up to 31 bits. The B bus operand specifies the shift distance. These two units are placed in parallel to reduce the levels of logic that must be traversed each cycle, but they do not operate in parallel.

An immediate operand may be used instead of one register source operand. The immediate operand comes from the instruction on a bus from the instruction unit, is extended to 32 bits, and put on B bus. A bit in the instruction is used to indicate whether an immediate operand or a register is used.

LOAD and STORE instructions require two instruction cycles for execution. In the first instruction cycle, the address is computed by adding two operands. Address calculation uses the same data path as is used for arithmetic operations. The actual LOAD or STORE is performed in the second cycle of the instruction. It is possible to store, in a register, the effective address used in a LOAD or STORE. After the address is computed in the ALU it is available to registers on C bus. The processor organization also allows one to take an address directly from one register and then to perform an add using this register. In this case, the address comes directly from A bus. These two types of addressing are known as pre-increment and post-increment, respectively. These are similar to the autodecrement and autoincrement addressing modes of the VAX [LeEc80] except the increment value is specified by the instruction and can be any value, positive or negative, in the PSP. Either of these addressing modes makes it easy to step through sequential data such as arrays. The purpose of allowing both types of increment is to allow easy implementation of a stack in memory.

The PSW, included in the PSW Area, is part of the data path because the PC can be used as an operand for relative addressing in LOAD, STORE, and BRANCH instructions. The return mask is stored in the MASK register. The PSW and MASK can be read and written using the data path; they do not have access to B bus. BRANCH instructions use the data path for computing the target address, which becomes the new PC. There is also access to BOSRs and the operand "zero" from A

bus. Fig. 4.1 does not show where the mask is used to "mask" the stack pop.

4.3. Instruction Unit and Data Bus

Fig. 4.2 shows the organization of the Instruction Unit and the Data (D) Bus. Fetch (F) bus is used to take the next instruction address from the PC to the memory. LOAD and STORE addresses come from C bus if normal or pre-increment instructions. Addresses in post-increment LOAD and STORE instructions come from A bus. STORE data are taken from A bus and LOAD data go to C bus. D bus is the path between the chip's pins and the Instruction Unit and data path. It is used for data, addresses, and instructions. The D Bus Select Unit consists of a multiplexer and temporary register for selecting a bus and holding data temporarily for proper timing with D bus.

Instructions, when fetched, come in on D bus and are stored in the Instruction Register (IR) in one-cycle instructions. In two-cycle instructions the instruction is stored in the Temporary Instruction Register (TIR) during the first cycle and moved to the IR in the second cycle. From the IR, the instruction decoding takes place, with the control signals distributed to their proper destination. The literal operand comes from the IR.

4.4. The PSW Area

The processor's data paths are 32 bits wide. In order to save the PSW on a stack without using more than one stack, a 29-bit PC is used. By limiting instructions to word (four-byte) boundaries, the two least significant bits of the PC are always zero and do not have to be stored. The 27 bits occupied by the PC leave five bits of a word unused. Four of these are used for the condition code: carry, overflow, zero, and sign. These bits can be set, when indicated, by any instruction based on the ALU or shifter output of the first cycle of the instruction. The fifth bit indicates whether interrupts are enabled or disabled. This word, containing the PC, condition code, and interrupt mask, is the PSW. Fig. 4.3 shows the PSW Area organization. Some instructions use the entire PSW (27 + 5 bits), while others use only the PC (27 bits) with the other bits zero. This is indi-

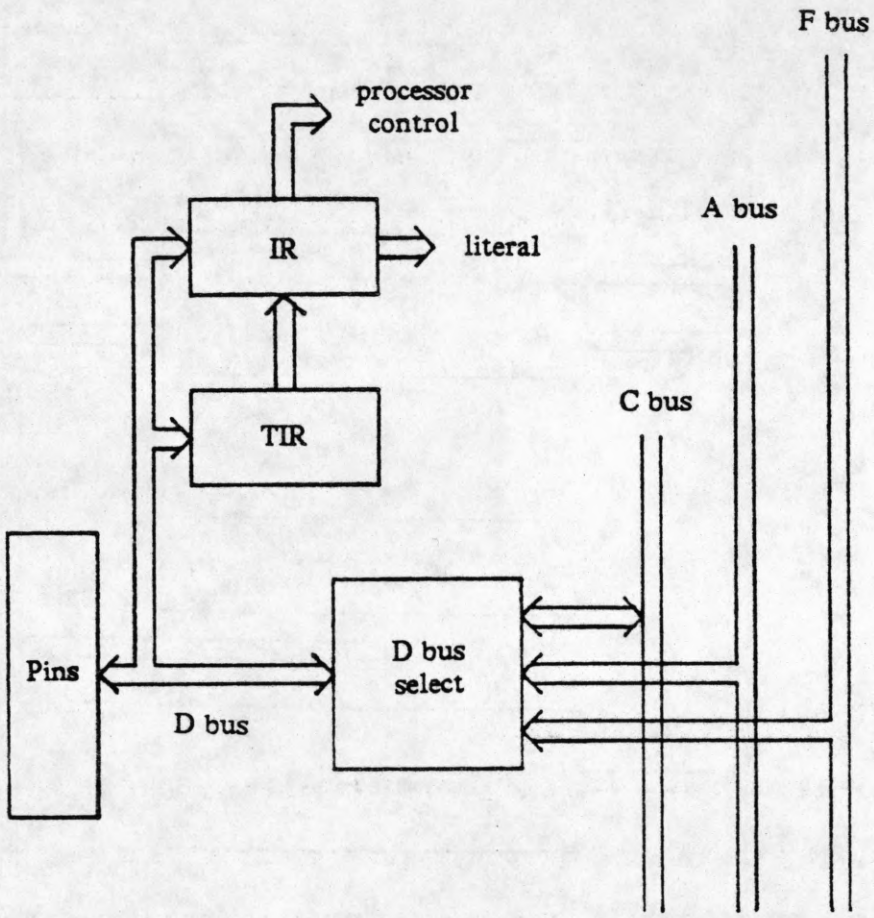

Fig. 4.2. Instruction Unit and Data Bus.

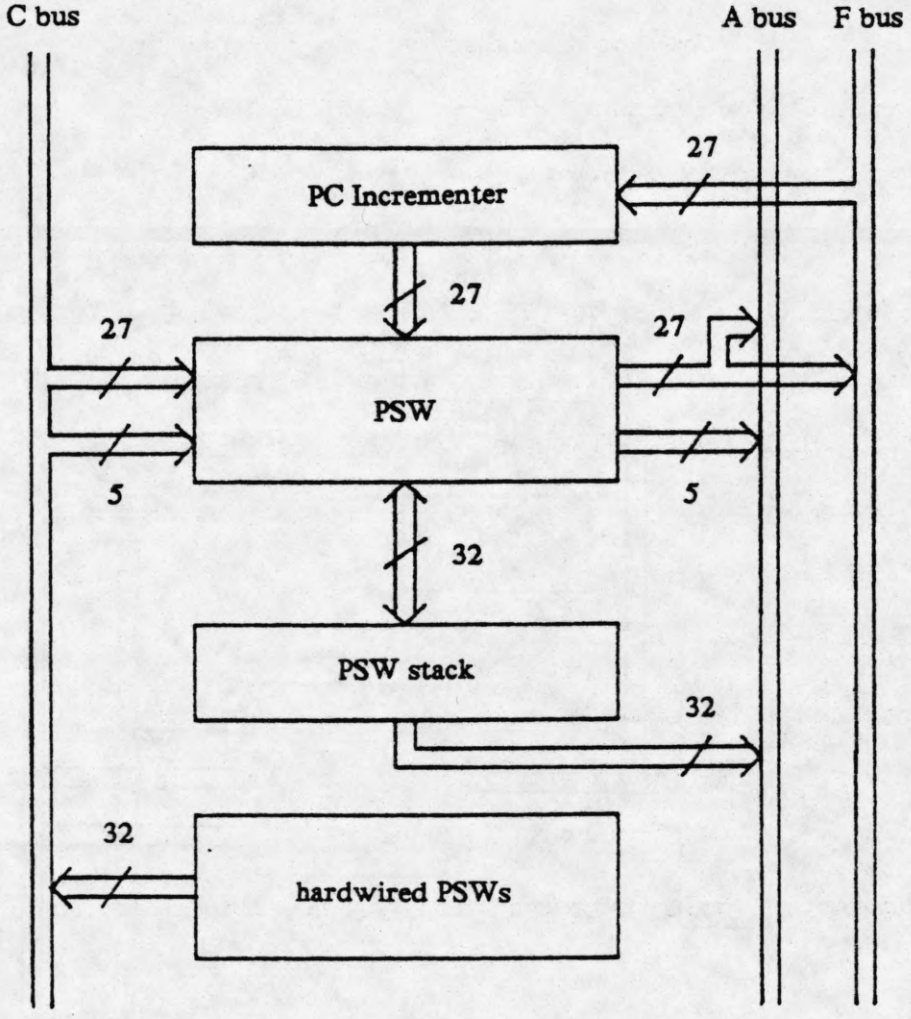

Fig. 4.3. PSW Area.

cated in the figure by the two paths going from the PSW to the busses and the numbers indicating the width of the busses. The PC has its own incrementer so that it can run in parallel with instruction execution. The PC contents are sent out as an address on F bus. The PC incrementer reads the PC from F bus, at that time, and stores the new PC later in the instruction cycle. The paths from the ALU and shifter to the condition code part of the PSW, the path by which a conditional branch reads the condition code, and the connections to the interrupt mask are not shown in the figure.

The PSW has its own stack, for saving state on procedure calls. As mentioned, this stack has two extra levels used for saving the PSW on an overflow. The interaction with the return mask is not shown. The overflow, underflow, and interrupt PSWs reach the PSW via C bus, which otherwise is not used at the time. These PSWs are hardwired into the processor; they contain the addresses of the routines and the interrupt mask set to disable interrupts. If desired, interrupts could be enabled during the interrupt routine, but this should not be done during overflow and underflow routines unless done with caution.

By including a separate bus from the PC to D bus and by giving the PC its own incrementer, an instruction fetch does not use any part of the data path. This allows instruction fetch to be overlapped with execution. For LOAD and STORE, a fetch occurs during the first cycle and the memory access during the second cycle. This results in one memory operation on every processor cycle.

Because the next instruction address must be computed explicitly, instructions which cause a transfer to another section of code cannot be done in one cycle. These include BRANCH, CALL, and RETURN instructions. At the time a BRANCH instruction is to begin execution the instruction statically following the branch is already being fetched and is in the IR at the end of the first cycle of the BRANCH instruction. A BRANCH that is not taken can then execute this instruction and continue, resulting in a one-cycle BRANCH instruction. A BRANCH that is taken would mean spending one cycle waiting for the target instruction to be fetched after its address is computed by

the BRANCH instruction. In order to avoid this delay, the instruction statically following the BRANCH is always executed independently of the result of the BRANCH. This is similar to the delayed jump of RISC [PaSe82]. Delayed CALL and RETURN are not possible because overflow and underflow require special handling.

CHAPTER 5

INSTRUCTION SET

5.1. Instruction Format

In this chapter, a description of the instruction set is given. Instructions, like data, are 32 bits wide. There is only one instruction format, but the fields are interpreted in different ways for some instructions. The instruction format is given in Fig. 5.1. The numbers in the figure indicate the bit positions within the word. The instruction format is divided into four fields: opcode, Rd, Rs1, and Rs2. The first field specifies the operation to be performed. Five bits of the opcode specify the basic instruction. The sixth bit, bit 26, specifies whether the condition code should be set.

In most instructions there are two source operands, Rs1 and Rs2, and one destination operand, Rd. Rd and Rs1, however, also have other uses in certain instructions. The first operand, Rd, is five bits and is normally the destination register. Bit 25 specifies how the other four bits are to be

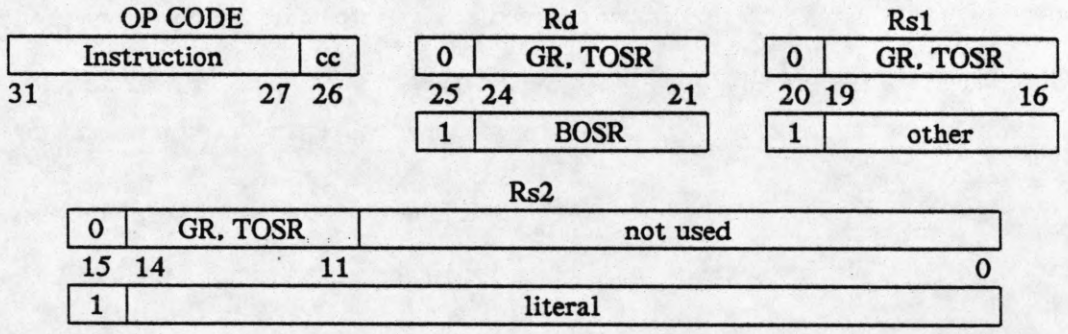

Fig. 5.1. Instruction format.

used. The destination field specifies one of 16 GRs and TOSRs or one of 9 BOSRs. However, if a BOSR is specified as a destination, no register is written. Rd is used as a source in STORE instructions. In STORE instructions it is useful to be able to specify a BOSR for overflow handling. The condition for branches is also specified by Rd. The second operand, Rs1, is also five bits and is normally used as a source register. The first bit, bit 20, specifies one of two interpretations of the other four bits. These bits specify one of 16 GRs and TOSRs or one of the following: PC, PSW, MASK, or ZERO (all bits are zero). It would also be possible to use these four as sources for STORE instructions in Rd. Rs1 serves as both a source and a destination in post- and pre-increment LOAD and STORE. Only GRs and TOSRs are allowed to be written, however. The third operand, Rs2, is either five or 16 bits and is used as a source. Bit 15 is used to specify whether four bits are used to select a register or a 15 bit immediate operand is used. When an immediate operand is used, it is extended to 32 bits using the most significant bit, bit 14, of the operand. If an undefined bit pattern is specified for Rd or Rs1, no word is read, resulting in an unknown state, or no word is written, depending on the use of the field.

5.2. Instruction Set

The instruction set is shown in Table 5.1. Memory addresses are specified in parentheses in the table. Most instructions read two registers, perform some operation on the two operands, and store the result in a register. With the encoded source information, the instruction set becomes very flexible. Instructions which modify special registers, such as PC or MASK, have the destination specified by the instruction opcode rather than encoded in the register field. This prevents such possibilities as allowing every instruction to modify the program counter. If one were to write in assembly language, one could create mnemonics for all the useful combinations of instructions and operands and let the assembler code them into the actual instructions. For example, a COMPARE instruction would be translated by an assembler to a SUBTRACT instruction specifying that the condition code be set and indicating in the destination coding that no register is to be written.

Table 5.1. Instruction Set.

Instruction		Description
ADD	Add	Rd ← Rs1 + Rs2
ADDC	With Carry	Rd ← Rs1 + Rs2 + c
SUB	Subtract	Rd ← Rs1 − Rs2
SUBC	With Carry	Rd ← Rs1 − Rs2 − c
SUBR	Subtract Reverse	Rd ← Rs2 − Rs1
SUBRC	With Carry	Rd ← Rs2 − Rs1 − c
AND	And	Rd ← Rs1 * Rs2
OR	Or	Rd ← Rs1 | Rs2
XOR	Exclusive Or	Rd ← Rs1 xor Rs2
SHL	Shift Logical	Rd ← Rs1 shifted by Rs2
SHLC	With Carry	Rd ← Rs1,c shifted by Rs2
SHA	Shift Arithmetic	Rd ← Rs1 shifted by Rs2
LD	Load	Rd ← (Rs1 + Rs2)
LDR	Pre-Increment Load	Rd ← (Rs1 ← Rs1 + Rs2)
LDO	Post-Increment Load	Rd ← (Rs1); Rs1 ← Rs1 + Rs2
ST	Store	(Rs1 + Rs2) ← Rd
STR	Pre-Increment Store	(Rs1 ← Rs1 + Rs2) ← Rd
STO	Post-Increment Store	(Rs1) ← Rd; Rs1 ← Rs1 + Rs2
BRC†	Branch On Condition	PC ← Rs1 + Rs2, Rd is condition
CALL	Call	push stacks; PC ← Rs1 + Rs2
RETURN	Return	MASK ← Rs1 + Rs2; pop stacks
PUTPSW†	Put PSW	PSW ← Rs1 + Rs2
OVRET†	Overflow Return	Mask ← Rs1 + Rs2; pop PSW stack
EI	Enable Interrupts	Set Interrupt flag; Rd ← Rs1 + Rs2
DI	Disable Interrupts	Clear Interrupt flag; Rd ← Rs1 + Rs2
CSLI	Clear SLI	Clear SLI; Rd ← Rs1 + Rs2

†This instruction uses delayed branch

Specifying no destination register could be used in any instruction where setting the condition code is the only desired result. When the Rs1 field specifies ZERO, a literal can be loaded into a register. Similarly, a register-to-register move can be performed. Moving the PC, PSW, or MASK to a register is done by specifying a literal with value zero in Rs2.

All instructions use either the ALU or shifter. The condition code is set based on the output of these units. For some instructions, such as Enable Interrupts, one would not want to do any data path operation. An operation is specified, however, to make the condition code setting defined in the same way for all instructions and to make all instructions similar. For some instructions,

this allows the possibility of doing useful work on an otherwise unused data path. The compiler can specify the operation if needed, rather than using the instruction decoder to specify that there is no data path operation for this instruction.

5.2.1. Arithmetic and Logical Instructions

The standard arithmetic and logical instructions are provided. Add, subtract, and reverse subtract are available, each also in a form using the carry condition code bit. The four-bit condition code is set, when indicated. It is up to the programmer to interpret the condition code setting with respect to signed or unsigned arithmetic. AND, OR, and EXCLUSIVE-OR are the available logical functions. The condition code can be set using these instructions; overflow and carry bits are cleared. Arithmetic and logical instructions execute in one cycle.

5.2.2. Shift Instructions

Arithmetic and logical shifting can be done in both directions. The distance and direction are specified by the six low order bits of Rs2. These bits are interpreted as a two's complement number. SHL shifts the bits the specified amount with zero filling each of the vacated bits. SHLC shifts the carry into the vacated bits. SHA shifts the sign bit in from the left on right shifts. The carry condition code is set when indicated by the bit that was originally in position zero, or thirty-one, depending on the direction of shift. The overflow condition code can be set only in an arithmetic left shift. Shift instructions require one cycle.

5.2.3. Load and Store Instructions

LD and ST compute the memory address by adding Rs1 and Rs2. The same computation happens in LDR and STR, but the address is also stored in Rs1, if Rs1 is a GR or TOSR. In LDO and STO, Rs1 and Rs2 are added and stored as above, but the address is taken from Rs1, before the addition takes place. PC-relative and absolute addressing are possible by selecting Rs1 as PC or ZERO. Indexed addressing is achieved by specifying a base address in a register and specifying a

displacement. Accessing a stack in memory is facilitated by the post- and pre-increment instructions. These instructions can also be used to step through an array, for example in a block move. BOSRs are stored when specified by Rd. LOAD and STORE each require two cycles. When specified, the condition code is set as a result of the address arithmetic.

5.2.4. Branch Instruction

BRC instruction specifies a condition on which to branch as the first operand, Rd. The condition can specify an unconditional branch or an unconditional no-branch as well as conditional branches. A standard four-bit condition specification, similar to the VAX [LeEc80], is used. The other two operands specify the address. If the condition is TRUE, the PC is changed to the result of adding Rs1 and Rs2. For a FALSE condition, the PC is incremented as in most other instructions. The addition of Rs1 and Rs2 also takes place in this case, but the results is not stored. As in a LOAD instruction, different addressing modes can be obtained using various source encodings. Because there can be two registers used to specify an address, a multiway branch is possible. The instruction statically following BRC is always executed so BRC can often effectively execute in one cycle. The instruction following a conditional branch should not use the PC because the value of the PC seen by that instruction is not known at compile time. The instruction can set the condition code, as the BRC would have already decided whether the branch is taken or not. An interrupt cannot occur between BRC and the following instruction. Because BRC can change the PC, while other instructions increment it, an interrupt would result in the wrong address being saved on the stack. BRC, therefore, ignores any interrupt.

5.2.5. Call and Return Instructions

CALL checks for overflow. If no overflow, the stacks are pushed, saving the return address. The new PC is obtained by adding Rs1 and Rs2. If there is an overflow, the PSW stack is pushed twice, saving the return address and the called procedure's address. A jump to the overflow rou-

tine is then made. The overflow indicator is cleared. Execution after the overflow routine resumes with the first instruction of the called routine. The condition code saved by CALL is the condition code at the time CALL begins. If CALL sets the condition code, based on the address computed, it is not saved, but passed to the called procedure. The CALL instruction takes two cycles if there is no overflow and three cycles if there is an overflow.

In RETURN, Rs1 and Rs2 are read to compute the mask which is then stored in the MASK register. RETURN checks for underflow. If there is none, the stacks are popped. If the PSW stack is masked, the PC is not changed. The next instruction fetch begins after the stack operation. If there is an underflow, the PSW stack is pushed. This is used in the underflow routine in the event that the PSW stack is masked. A jump to the underflow routine is executed and the underflow indicator is cleared. RETURN requires two cycles for execution if there is no underflow and three cycles if there is an underflow.

The return mask is specified by RETURN. In most cases it would be specified as an immediate operand. The nine low order bits are used and stored in the nine-bit MASK register. Each of the eight register stacks and the PSW stack is associated with one mask bit. CALL can specify the next sequential address using PC-relative addressing, effectively executing a stack push without a jump. The PSW stack mask provides the inverse of that operation: a stack pop without a jump.

5.2.6. Miscellaneous Instructions

A few more instructions are needed. In order to copy the MASK or PSW to a register, an arithmetic instruction can be used with the proper source encoding. Reading the MASK is necessary in the underflow handling routine. The PUTPSW instruction is used to set the entire PSW. OVRET is used to resume processing after a stack overflow. It pops the PSW stack using only one bit of the mask. Unlike RETURN this mask is not saved in the MASK register. A PSW stack pop cannot cause an underflow because the underflow indicator is only changed by popping the register stacks. Both PUTPSW and OVRET use the delayed branch. They also must ignore interrupts. A

pair of instructions is included for enabling and disabling interrupts. The new interrupt mask bit is set by these instructions, but it does not go into effect until the next instruction. One additional instruction is required for implementing process swaps. The SLI is cleared by this instruction. Each of the miscellaneous instructions requires one cycle.

5.3. Example Code for CALL Overflow and RETURN Underflow

Figures 5.2 and 5.3 show examples using PSP assembly code. The overflow and underflow routines are given. The column labeled "Cycle" counts the number of instruction cycles used in the routines at the time the instruction is to begin. In the underflow routine, each bit of the mask is checked to determine whether to load data or to skip over data. For the underflow routine, the worst case of no registers being masked is assumed and is indicated in the number of cycles. Hardwired addresses specify the beginning addresses of the routines, but only two instructions are

Cycle	Instruction	Comment
(0)	CALL	Instruction causing overflow
(3)	†BRC Oflow	These two instructions located
(4)	ST R1,temp	at hardwired address
(6)	Oflow: LD R1,OSP	Load Overflow Stack Pointer
(8)	STO BOSPSW,R1,−4	Store BOSPSW using post-increment
(10)	STO BOSR15,R1,−4	Store each BOSR
(12)	STO BOSR14,R1,−4	
	. . .	
(24)	STO BOSR8,R1,−4	
(26)	CALL PC,0	Push stacks (PC ← PC + 0)
(28)	†OVRET,mask	Pop PSW stack, but increment PC
(29)	ST R1,OSP	Store new OSP
(31)	†OVRET	Pop PSW stack, PSW ← call addr
(32)	LD R1,temp	Restore R1
(34)	first instruction of called routine	

†Uses delayed branch

Fig. 5.2. Call overflow routine.

Cycle	Instruction	Comment
(0)	RETURN	Instruction causing underflow
(3)	†BRC Uflow	These two instructions located
(4)	ST R1,temp	at hardwired address
(6)	Uflow: LD R1,OSP	Load Overflow Stack Pointer
(8)	ST R2,temp2	
(10)	SHA,cc R2,MASK,Right1	Check mask bit for R8
(11)	†BRC if carry=0 A1	(R8 mask bit)
(12)	SHA,cc R2,R2,Right1	Check mask bit for R9
(13)	†BRC L1	
(14)	LDR R8,R1,4	Load R8, if mask bit was 1
	A1: ADD R1,R1,4	Update OSP if no load
(16)	L1: †BRC if carry=0 A2	(R9 mask bit)
(17)	SHA,cc R2,R2,Right1	Check mask bit for R10
(18)	†BRC L2	
(19)	LDR R9,R1,4	Load R9
	A2: ADD R1,R1,4	
	. . .	
(46)	L7: †BRC if carry=0 A8	(R15 mask bit)
(47)	SHA,cc R2,R2,Right1	Check mask bit for PSW
(48)	†BRC L8	
(49)	LDR R15,R1,4	Load R15
	A8: ADD R1,R1,4	
(51)	L8: LDR R2,R1,4	R2 ← new PSW
(53)	ST R1,OSP	Store new OSP
(55)	†BRC if carry=0 L9	(PSW mask bit)
(56)	LD R1,temp	Restore R1
(58)	†PUTPSW R2,0	PSW ← R2
(59)	LD R2,temp2	Restore R2
(61)	execute instruction in procedure returned to	
(58)	L9: †OVRET	Pop PSW stack
(59)	LD R2,temp2	Restore R2
(61)	instruction following return (PSW stack was masked)	

†Uses delayed branch

Fig. 5.3. Return underflow routine.

allowed at each address, so one must be a branch. The remainders of the routines are located elsewhere in memory. The first instruction listed for each routine is the last instruction from the executing program. During the routines, one or two registers are needed. The registers affected are temporarily stored in system-reserved locations. Instructions which use the delayed branch are marked in the figures. Post- and pre-increment instructions are used for accessing the overflow stack. This stack, in memory, is a separate stack from the *call frame* stack. This stack is referenced with an overflow stack pointer (OSP). The underflow routine has two possible endings depending on the PSW stack mask bit.

CHAPTER 6

IMPLEMENTATION

6.1. Introduction

This chapter presents more detail of the PSP implementation than is given in the preceding chapters. The instruction set is presented, again, with a detailed description of events that take place, including timing information. This should clarify any unclear points from Chapter 5. Some nMOS implementation issues are presented including a floor plan and a logic design for a stack cell.

6.2. Instruction Set Detail and Timing

The instruction set is examined in this section. This time, each instruction is described with timing information and more detail about the actual flow of information through the processor. Fig. 6.1 is a list of all instructions and their detailed description. Each is broken into its clock phases. The processor uses a three-phase clock; that is, for each instruction cycle, there are three machine cycles. In general, registers are read in phase one, manipulated in phase two, and the result is stored in phase three. For multi-cycle instructions there are more than three phases. Some instructions are given two sequences of events. The particular sequence depends on some condition encountered during execution such as a stack overflow. In instances where there appear to be two WRITEs to the same location in the same clock phase, the events can be considered to take place in the order listed. The figure's "KEY" lists a number of types of events. These are used as a short-hand notation in the descriptions. Register READs made to D bus result in a word sent off-chip. Data coming from off-chip results in a register WRITE from D bus. A memory register (MR) is located in the D Bus Select unit to temporarily hold data.

KEY

Read(r , b): read register r to bus b
Execute(op): A and B bus operands to ALU or shifter;
perform operation op ; result to C bus
Write(r , b): write register r from bus b
RTransfer(r 1, r 2): transfer from register r 1 to register r 2
BTransfer(b 1, b 2): transfer from bus b 1 to bus b 2
Fetch: read PC to F bus, then to incrementer and D bus
CC: if indicated in instruction, set condition code
Intr: if interrupts are enabled, check for interrupts
Incr: if no interrupt, write to PC from incrementer
Push: push all stacks
Pop: pop all stacks using mask
PushPSW: push PSW stack
PopPSW: pop PSW stack using mask
SLI(b): check SLI overflow or underflow bit b
Clear(i): clear indicator i
IM(m): change interrupt mask to m

INSTRUCTIONS

Arithmetic, Logical, and Shift instructions
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(opcode)
3-CC, Write(Rd,C), Intr, Incr, Write(IR,D)

Load instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(MR,C), Intr, Incr, Write(TIR,D)
4-Read(MR,D)
6-BTransfer(D,C), Write(Rd,C), RTransfer(TIR,IR)

Pre-Increment Load instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(Rs1,C), Write(MR,C), Intr, Incr, Write(TIR,D)
4-Read(MR,D)
6-BTransfer(D,C), Write(Rd,C), RTransfer(TIR,IR)

Post-Increment Load instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add), Write(MR,A)
3-CC, Write(Rs1,C), Intr, Incr, Write(TIR,D)
4-Read(MR,D)
6-BTransfer(D,C), Write(Rd,C), RTransfer(TIR,IR)

Fig. 6.1. Instruction set detail.

Store instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(MR,C), Intr, Incr, Write(TIR,D)
4-Read(MR,D), Read(Rd,A), Write(MR,A)
5-Read(MR,D)
6-RTransfer(TIR,IR)
Pre-Increment Store instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(Rs1,C), Write(MR,C), Intr, Incr, Write(TIR,D)
4-Read(MR,D), Read(Rd,A), Write(MR,A)
5-Read(MR,D)
6-RTransfer(TIR,IR)
Post-Increment Store instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add), Write(MR,A)
3-CC, Write(Rs1,C), Intr, Incr, Write(TIR,D)
4-Read(MR,D), Read(Rd,A), Write(MR,A)
5-Read(MR,D)
6-RTransfer(TIR,IR)
Branch instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add), evaluate condition
3-CC, Write(PC,C) or Incr, Write(IR,D)
Call instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add), SLI(Overflow), if overflow: O2
2(cont.)-Push
3-CC, Write(PC,C), Write(TIR,D)
4-Fetch
6-Intr, Incr, Write(IR,D)
O2-PushPSW
O3-CC, Write(PC,C), Write(TIR,D)
O5-PushPSW
O6-Read(OverflowPSW,C), Write(PSW,C), Clear(Overflow)
O7-Fetch
O9-Intr, Incr, Write(IR,D)

Fig. 6.1. (cont.)

Return instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add), SLI(Underflow), if underflow: U3
3-CC, Write(MASK,C), Pop, Write(TIR,D)
4-Fetch
6-Intr, Incr, Write(IR,D)
U3-CC, Write(MASK,C), Write(TIR,D)
U5-PushPSW
U6-Read(UnderflowPSW), Write(PSW,C), Clear(Underflow)
U7-Fetch
U9-Intr, Incr, Write(IR,D)
Put PSW instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(PSW,C), Write(IR,D)
Overflow Return instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Incr, PopPSW, Write(IR,D)
Enable Interrupts instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(Rd,C), Intr, Incr, IM(Enable), Write(IR,D)
Disable Interrupts instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(Rd,C), Intr, Incr, IM(Disable), Write(IR,D)
Clear Stack Level Indicator instruction
1-Read(Rs1,A), Read(Rs2,B), Fetch
2-Execute(add)
3-CC, Write(Rd,C), Intr, Incr, Clear(SLI), Write(IR,D)
Interrupt
(last instruction)-Intr, Incr, etc.
2-SLI(Overflow), if overflow: O2
2(cont.)-Push
3-Read(InterruptPSW,C), Write(PSW,C), Clear(Interrupt)
4-Fetch
6-Intr, Incr, Write(IR,D)
O2-PushPSW
O3-Read(InterruptPSW,C), Write(PSW,C), Clear(Interrupt)
O5-PushPSW
O6-Read(OverflowPSW,C), Write(PSW, C), Clear(Overflow)
O7-Fetch
O9-Intr, Incr, Write(IR,D)

Fig. 6.1. (cont.)

The first phase of every instruction is identical. This allows an instruction to begin before the opcode has been decoded. Only the sources for busses A and B need to be decoded at this time. These are completely specified by operands Rs1 and Rs2. The ALU or shifter operation is executed on these operands in phase two. The result is stored in a register in phase three. If the condition code bit in the instruction was set, the condition code is set, also in phase three.

The instruction fetch begins in phase one of every instruction. The current value of the PC is read to F bus, the incrementer, and D bus. When the instruction word arrives from memory in phase three it is transferred to the IR or TIR. IR is used when a one-cycle instruction is executing. Two-cycle instructions use TIR. CALL and RETURN, which explicitly change the PC and do not use a delayed branch, make two instruction fetches. The first is to TIR and is ignored. The second fetch is to IR. Other two-cycle instructions move the contents of TIR to IR in phase six. In general the PC receives its incremented value in phase three. Interrupts are also checked in phase three. The presence of an interrupt prevents the PC from being incremented. This PC will be saved on the stack and the instruction will be refetched and executed after interrupt handling. Interrupt timing is similar to CALL and is given in Fig. 6.1. In LOAD and STORE instructions, the incremented PC and the presence of an interrupt are remembered through the second cycle. Since the next instruction is fetched and PC incremented in the first cycle, and incrementing should not occur if there is an interrupt, interrupts must be checked in the first cycle for these instructions.

6.3. Data Path

The logical organization of the PSP has been described previously. In this section, some of the details of the physical organization of the chip will be discussed. Throughout this section it will be assumed that the chip is to be implemented in nMOS technology, although certain details may be somewhat independent of the choice of technology. Fig. 6.2 shows a sketch of the floor plan of the processor. The drawing is not to scale, but is used to show the relative positions of each unit. The data path, the right half of the figure, is organized as 32 one-bit slices. A and C busses run

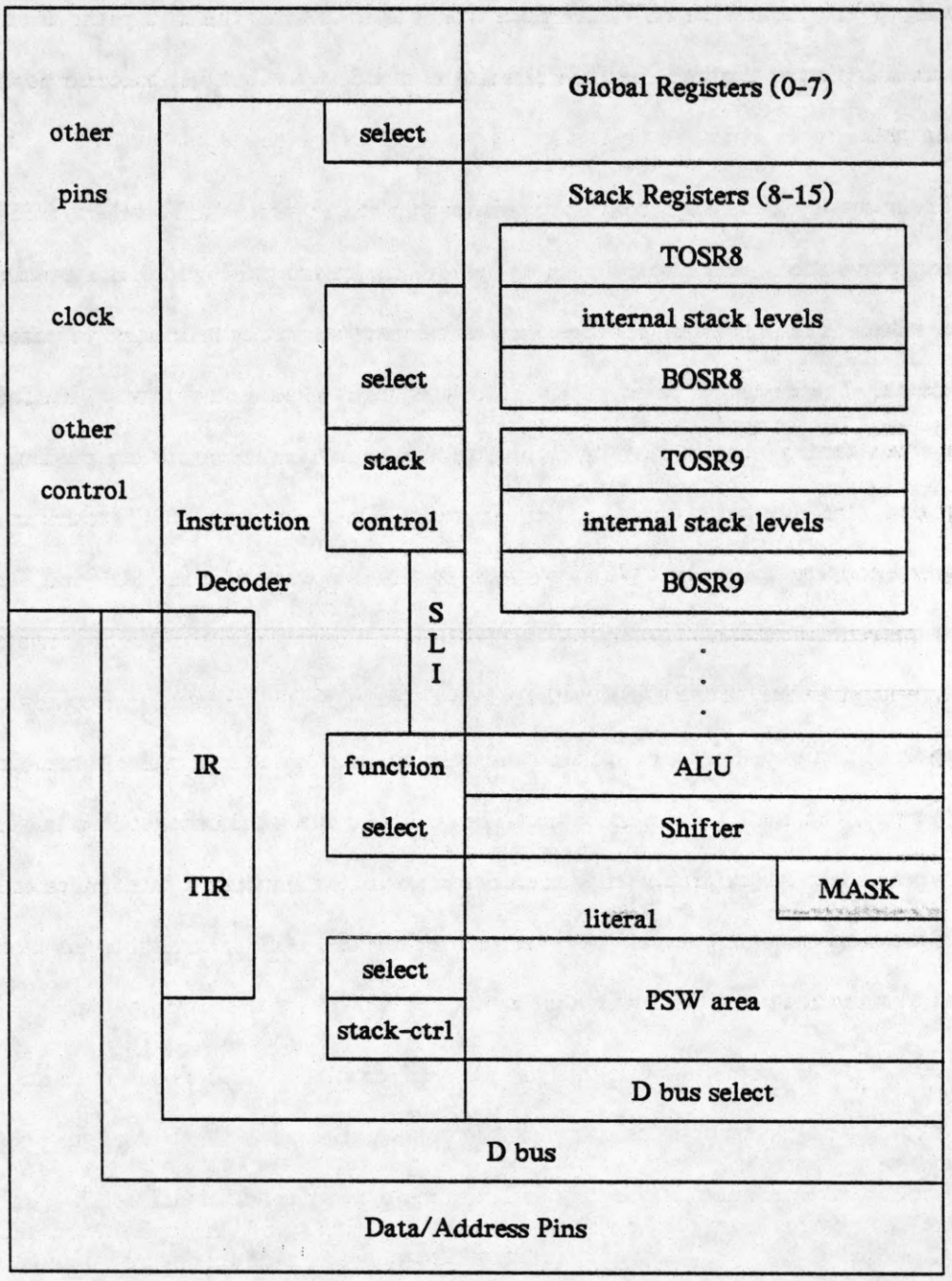

Fig. 6.2. Chip floor plan.

vertically the entire length of the data path. B bus runs vertically from one end, the General Registers, to just before the PSW, the place where literals enter the data path. F bus begins where B bus ends and runs through the PSW Area to D Bus Select. Control lines run horizontally across the data path.

The necessary controls for registers are register READ and WRITE select, BOSR READ select, push and pop signals, and clock signals to refresh the registers. Fig. 6.3 is a possible design for a one bit stack. The stack cell can be repeated as many times as necessary to achieve the desired stack depth. The design is based on the stack design in Mead and Conway [MeCo80]. The clock phases when control signals may be active are shown in the figure. When pushing the stack, the "push" and "trd" (transfer down) signals are used. The "pop" and "tru" (transfer up) signals are used when popping the stack. When no push or pop is occurring, the "trd" and "tru" signals are used to refresh the stacks. The "mask" signal does not come from the MASK register but directly from one bit of C bus. The MASK register is written at that time, but is necessary only for a possible stack underflow. Registers are read in phase one and written in phase three. In Fig. 6.3, data move vertically in stack pushes and pops. The stacks lie along the busses. This is shown in Fig. 6.2 with two register stacks. Other register stacks are oriented similarly. The length of the chip register file, therefore, is a function of the total number of GRs, TOSRs, BOSRs, levels of stack per register, and number of stacks. The width is 32 bits.

Mead and Conway [MeCo80] also give designs for an ALU and a barrel shifter. Modified versions of these designs could be used in the PSP. The ALU is general purpose; only some of the possible functions are necessary. The shifter design requires some additional hardware for the instruction set of the PSP. The Mead and Conway design is for a double word, unidirectional shifter. This can easily be changed to a single word, bidirectional shifter by putting the word to be shifted in the proper half of the double word. The bit to be shifted in is put in the other half of the double word. The logic needed to set the condition code must also be added for the ALU and the shifter.

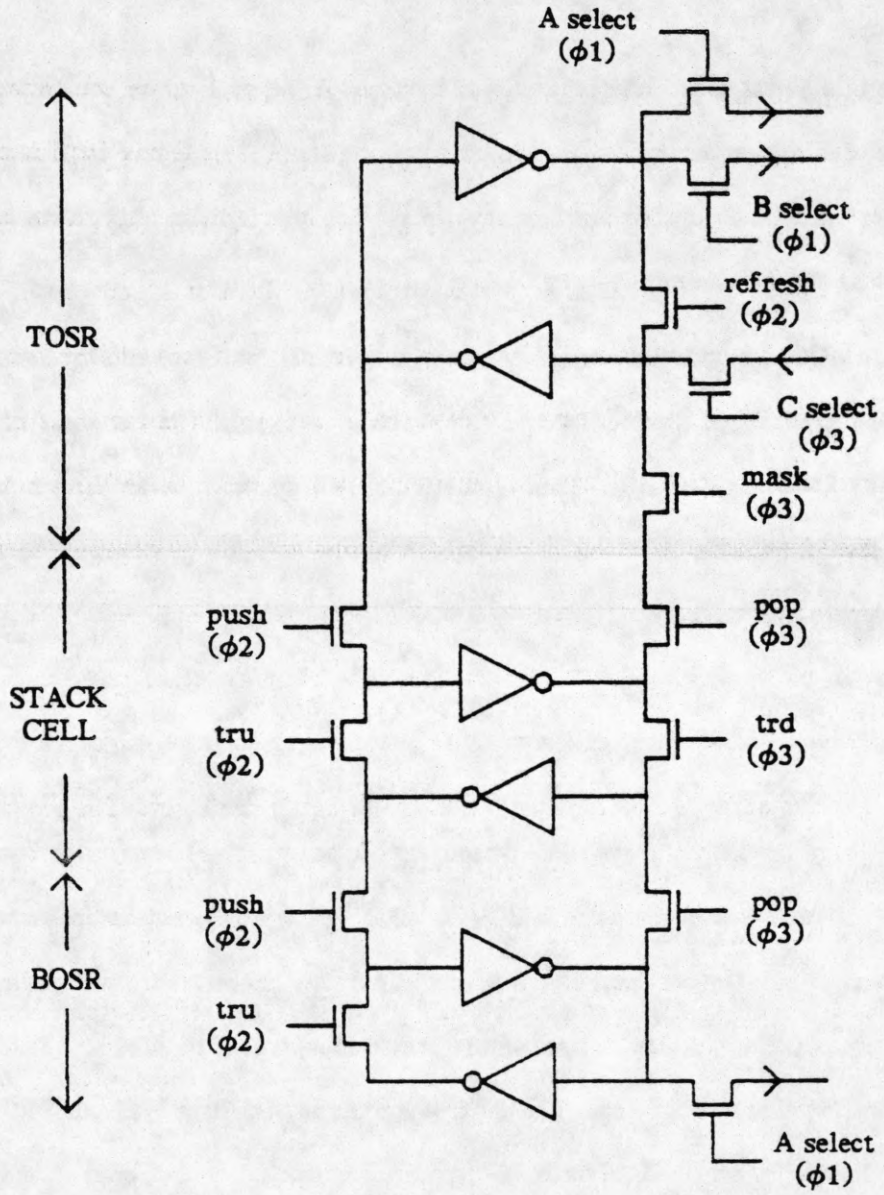

Fig. 6.3. A one-bit register stack design of depth three.

The PSW is similar to a stack register with some additional connections. READ and WRITE must be possible for the 27-bit PC and for the full 32-bit PSW. An additional path is required from the incrementer. The condition code and interrupt mask must also operate independently of the PC.

The D Bus Select Unit selects one of the busses A, C, or F to be connected to D bus. It also contains a buffer to temporarily store the data coming from A or C bus until it can be put on D bus at the proper time. The buffer can be very simple because it need only store data for one or two clock phases as the data need not be refreshed before it is sent out.

In an nMOS design, because of the unsymmetrical pull-up and pull-down times, certain busses can be precharged during an idle clock phase so that the lower time of pull-down can be achieved for all cases [MeCo80]. This is particularly important with long busses, as in the PSP. The ALU design includes a carry chain which could also be precharged to minimize the time required to propagate the carry.

6.4. Control Path

The control section of the PSP occupies the left half of Fig. 6.2. Due to the small instruction set and the large on-chip storage, one would expect the chip area used for control to be a small fraction of the total chip area as in RISC [Fitz82]. The control section consists of the instruction registers, instruction decoder, and control for the data path. Instruction decoding may begin when the IR is loaded in the last clock phase of the previous instruction. Operand fields are sent to register selectors for phase one READs. The control signals for the first cycle should all be generated by the end of phase one. In the data path, each possible action can occur only during one specific clock phase. Timing information, therefore, need not be added to the control signals until the signals are close to the data path. The signals for subsequent cycles of multi-cycle instructions can be generated at a later time in their own decoders.

For CALL and RETURN, where there are two possible operations to be performed in the first cycle, depending on overflow or underflow, some additional information is needed. It can be noted, however, that except for the stack operation itself, the first cycle of the CALL is not dependent on overflow information. A RETURN is similar. In CALL, for example, the instruction decoder could send out commands for a push of all stacks and a PSW stack push. The overflow bit is also input to the stack controller and determines which command is honored. The instruction decoder does not need overflow or underflow information until second-cycle decoding begins. The stack control itself generates the appropriate signals to push, pop, or refresh stacks elements depending on the instruction.

The SLI can be implemented as a one-bit stack in parallel with the other stacks. The value of each bit indicates whether the data at that level are valid or not. This indicator should be cleared when the processor is reset. A stack push shifts this stack with a 1 entering at the top. On a pop, a 0 can be shifted in at the bottom. The overflow indicator used by the call instruction is the bit corresponding to the BOSRs. It is read before the push signal is generated. The underflow signal is obtained from the bit corresponding to the level below the TOSRs (the TOSRs always have valid data). The appropriate bit is reset by CALL or RETURN after the overflow or underflow has been detected. With this implementation of the SLI, however, the underflow bit need not be changed. The overflow bit must be reset so that a CALL within the overflow routine will not cause an overflow.

The registers for A and C busses are specified by Rs1 and Rd, respectively, except for a few cases. In LDR, LDO, STR, and STO, Rs1 is used both as a source and a destination so it specifies registers for both A and C busses. Likewise in STORE instructions, Rd is used as a source. Because of this and because A and C busses are active at different times, it is possible to use a single register decoder for both busses. Temporary storage is needed in the register selector to help with the timing and to save register addresses that are used twice. In addition to the operand specifications,

register selectors need to know which particular type of LOAD or STORE instruction is being executed.

D bus is shown in Fig. 6.2. It is the common path used by the data path and instruction decoder to communicate with the external world. Also shown in the figure is space for "other" control. This includes clock signals, power supply input, and other necessary control of the processor.

CHAPTER 7

CONCLUSIONS

7.1. Summary of Results

Due to the large overhead of saving processor registers and passing parameters in a procedure call, a method of reducing this overhead can greatly improve overall processor performance. The Parallel Stack Processor was studied for this purpose. Parallel stacks are used to save some of the processor registers and the PSW when calling a procedure or when processing an interrupt. A standard processor must use memory to save the processor's status. When the register stacks overflow, one level of data is moved to memory. The PSP allows parameters to be passed in registers because there is a complete overlap of registers between the calling and the called procedure. A mask is used on a RETURN to distinguish between *call-by-value* and *call-by-reference* parameters. The mask determines whether a change is made to the value of a parameter, as seen by the calling procedure. The problem of taking addresses of register-allocated variables when passing *call-by-reference* parameters is eliminated by using the return mask. High-level language use of the PSP was also studied. The PSP is particularly suited for languages that are not block structured because in these languages there is no need to access intermediate levels of the stack. The parallel stack architecture allows a large amount of storage to be accessed with a small address.

A performance model was developed to predict the performance of the PSP and to compare PSP with a standard processor. Two different strategies of variable allocation and parameter passing were used on both the PSP and a conventional processor. It was found that significant performance improvements on the order of 25 to 30 percent, due to reduction of memory references, are possible with the parallel stack architecture. From the performance results, the PSP should have 8

register stacks each of depth 8.

A simple three bus processor architecture incorporating the parallel stack architecture was proposed. PSP is a reduced instruction set computer, as it has a small instruction set and most instructions take one cycle to execute. Instruction fetch and execution are overlapped. The branch problem in pipelined machines is alleviated by always executing the first instruction following the BRANCH in the code. LOAD and STORE instructions are available that update the address register in addition to performing the memory operation. Instruction operands are coded to allow flexibility in selecting operands. This allows a variety of addressing modes while keeping the instruction set simple.

Finally, some implementation issues for this particular processor were discussed. This included detailed instruction timing and some thoughts about the floor plan and physical organization of the PSP.

7.2. Suggestions for Further Research

A number of different areas for further research are suggested by the parallel stack architecture. Several factors were left out of the performance model. With the availability of more statistics, fewer assumptions would have to be made. Particularly useful would be a distribution of the number of times that each variable is referenced. This could lead to using registers for the most frequently referenced variables. The arrangement of parameters before a procedure call begins would be useful to improve the model. This would lead to a better understanding of how parameters must be rearranged at a procedure call.

In order to get much more detailed performance information, it may be necessary to use simulation or to build the processor. These methods require assembly language versions of many benchmark programs to produce results that are valid for determining performance of a general computer system. Hand-coded programs could be used at first, but eventually a compiler would be desired. One problem for study is how to write a compiler for the PSP. Is it difficult to pass

parameters in registers between procedures? Variable rearrangement on procedure calls and register allocation are related. The order of the parameters in a procedure listing could affect performance if that specified how parameters were allocated. Is there a good way to specify to the calling procedure where to pass parameters for the best performance for the particular called procedure? One must also specify if some variables may be aliased. To make full use of the parallel stack architecture, register allocation and parameter passing should be used. However, the results from the performance model showed that using the stacks simply to save registers could achieve good performance gains. A simple compiler could use this approach.

The ideas of multiple register sets and reduced instruction set computers have been studied and will continue to be studied. Most of these studies have been in Pascal-like languages running non-numerical programs on small processors. In the work presented here, the performance measured the gain due to parallel stacks alone. Perhaps the chip area could have been better utilized with something other than stack, for example a cache. This may be true especially in numeric problems where the number of procedure calls is small. However, large computers, running non-numeric code, can probably see a savings in execution time with parallel stacks. Other types of languages could also be studied. In addition to procedure calls, parallel stacks might be used to save variables that will not be referenced for a certain period of time, even if no procedure call occurs. This would save memory references and not require parameter passing overhead.

This paper has presented an architecture with a potential for a good performance improvement over conventional architectures. Because of this, it would be beneficial to extend the parallel stack concept and to improve it further.

REFERENCES

[AhUl77] Aho, Alfred V., and Jeffrey D. Ullman, *Principles of Compiler Design*, Reading, MA: Addison-Wesley, 1977.

[Chai82] Chaitin, G. J., "Register Allocation & Spilling via Graph Coloring," *SIGPLAN Notices*, vol. 17, no. 6, pp. 98-105, June 1982.

[ClSt80] Clark, Douglas W., and William D. Strecker, "Comments on 'The Case for the Reduced Instruction Set Computer,' by Patterson and Ditzel," *Computer Architecture News*, vol. 8, no. 6, pp. 34-38, October 1980.

[Colw83] Colwell, Robert P., et al., "Peering Through the RISC/CISC Fog: An Outline of Research," *Computer Architecture News*, vol. 11, no. 1, pp. 44-50, March 1983.

[Dann79] Dannenberg, Roger B., "An Architecture with Many Operand Registers to Efficiently Execute Block-Structured Languages," *Proc. 6th Annual Symposium on Computer Architecture*, April 1979, pp. 50-57.

[DiMc82] Ditzel, David R., and H. R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 48-56.

[DiPa80] Ditzel, David R., and David A. Patterson, "Retrospective on High Level Language Computer Architecture," *Proc. 7th Annual International Symposium on Computer Architecture*, May 1980, pp. 97-104.

[Fitz82] Fitzpatrick, Daniel T., et al., "A RISCy Approach to VLSI," *Computer Architecture News*, vol. 10. no. 1, pp. 28-32, March 1982.

[HaKe80] Halbert, D., and P. Kessler, "Windows of Overlapping Registers," CS292R Final Project Reports, University of California, Berkeley, CA, June 1980.

[Hill83] Hill, Dwight D., "An Analysis of C Machine Support for Other Block Structured Languages," *Computer Architecture News*, vol. 11, no. 4, pp. 6-16, September 1983.

[JeWi74] Jensen, Kathleen, and Niklaus Wirth, *PASCAL User Manual and Report*, New York, NY: Springer-Verlag, 1974.

[Kate83] Katevenis, Manolis G. H., "Reduced Instruction Set Computer Architectures for VLSI," Report no. UCB/CSD 83/141, University of California, Berkeley, CA, 1983.

[KeRi78] Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

[Lamp82] Lampson, Butler W., "Fast Procedure Calls," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982, pp. 66-76.

[LeEc80] Levy, Henry M., and Richard H. Eckhouse, Jr., *Computer Programming and Architecture—The VAX-11*, Bedford, MA: Digital Press, 1980.

[MeCo80] Mead, Carver, and Lynn Conway, *Introduction to VLSI Systems*, Reading, MA: Addison-Wesley, 1980.

[Patt85] Patterson, David A., "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8-21, January 1985.

[PaDi80] Patterson, David A., and David R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News*, vol. 8, no. 6, pp. 25-33, October 1980.

[PaSe82] Patterson, David A., and Carlo H. Séquin, "A VLSI RISC," *Computer*, vol. 15, no. 9, pp. 8-21, September 1982.

[Radi83] Radin, George, "The 801 Minicomputer," *IBM Journal of Research and Development*, vol. 27, no. 3, pp. 237-246, May 1983.

[Site79] Sites, Richard L., "How to Use 1000 Registers," *Caltech Conference on VLSI*, January 1979, pp. 527-532.

[Tane74] Tanenbaum, Andrew S., "A Programming Language for Writing Operating Systems," Rep. IR-3, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1974.

[Tane78] _____, "Implications of Structured Programming for Machine Architecture," *Communications of the ACM*, vol. 21, no. 3, pp. 237-246, March 1978.

[Weic84] Weicker, Reinhold P., "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013-1030, October 1984.