REDUCED REGISTER SAVING/RESTORING IN SINGLE-WINDOW REGISTER FILES

Tomas Lang
Miquel Huguet

December 1985
CSD-850040

Reduced Register Saving/Restoring in Single-Window Register Files[†]

Tomás Lang and Miquel Huguet[††]

UCLA Computer Science Department
University of California, Los Angeles
Ca 90024

ABSTRACT

In a register-oriented architecture, the sequence of instructions corresponding to the function call construct has been identified as consuming a significant fraction of the execution time of typical programs written in high-level languages such as C. A large reduction in this time is obtained by the use of multiple-window register files. However, this approach has as disadvantages the cost of the large number of registers (in chips in a MSI/LSI implementation and in area in the VLSI case), the increase in cycle time in a VLSI implementation due to longer data buses, and the increase of the time for context switching. These disadvantages can be reduced by the use of multi-size windows and the implementation of the file as a set of shift registers. However, due to these disadvantages, the multiple-window approach might not be suitable for all situations. Consequently, it is interesting to develop architectures and compilers that reduce the cost of function call for the single-window case. In this paper we present schemes that permit the reduction of the memory traffic due to register saving and restoring, which is one of the major components of the execution time of function call/return. We also consider the implementation of the selected scheme and conclude that the implementation is feasible and might be a good use of resources to increase execution speed.

[†]This paper has been submitted for presentation at the *13th Annual International Symposium on Computer Architecture* (June 1986).

[††]This research is partially supported by a scholarship from CIRIT (*Generalitat de Catalunya*).

1. Introduction

In a register-oriented architecture, the sequence of instructions corresponding to the function call construct has been identified as consuming a significant fraction of the execution time of typical programs written in high-level languages such as C [LUND77, PATT82]. A large reduction in this time is obtained by the use of multiple-window register files [PATT82]. However, this approach has as disadvantages the cost of the large number of registers (in chips in a MSI/LSI implementation and in area in the VLSI case), the increase in cycle time in a VLSI implementation due to longer data buses, and the increase of the time for context switching [HENN84]. These disadvantages can be reduced by the use of multi-size windows [KATE83, HUGU85a, FURH85] and the implementation of the file as a set of shift registers [HUGU85b]. However, due to these disadvantages, the multiple-window approach might not be suitable for all situations. Consequently, it is interesting to develop architectures and compilers that reduce the cost of function call for the single-window case.

In this paper we present schemes that permit the reduction of the memory traffic due to register saving and restoring, which is one of the major components of the execution time of function call/return. We performed measurements on the programs described bellow, and determined that for one of the proposed schemes the traffic is reduced by 86% with respect to the conventional scheme of saving during the call the registers defined by the callee and restoring them upon return. We also consider the implementation of the selected scheme and conclude that the implementation is feasible and might be a good use of resources to increase execution speed.

Cost of Function Call/Return

The execution time of function call/return can be measured in terms of the number of instructions that have to be executed and of the data memory traffic involved. The subfunctions that contribute to this time are [HITC85]: i) the passing of parameters, ii) the saving and restoring of environment registers, iii) the saving and restoring of registers for local simple variables and temporaries preserved across function calls, iv) the creation and destruction of a new activation record, and v) the access in memory of those local variables that are not allocated to registers.

To reduce the execution time of the function call/return all these components should be considered. In this paper we concentrate on the saving and restoring of registers for local simple variables, which is a major component of the time [PATT82].

Measurements for Performance Evaluation

To evaluate the performance of the schemes proposed we have measured three relatively large programs: the UNIX* text formatting program (NROFF), processing one third of the FORTRAN 77 manual; the UNIX sorting program (SORT), sorting 2250 numbers; and the Portable C Compiler for the VAX-11** (VPCC), compiling one of its machine independent modules (*allo.c*). These include 500 functions and their execution has more than 1,500,000 function calls. We also report measurements for the benchmarks that have been used in several other studies [PATT82a, HITC85]; however, from the results reported here and in previous publications [HUGU85] it seems clear that the conclusions obtained from

*UNIX is a trademark of AT&T Bell Laboratories.

**VAX is a trademark of Digital Equipment Corporation these small programs are not very significant. We include the results for these small benchmarks in Appendix II.

The Compiler and the Number of Registers

The number of registers saved and restored in function calls is dependent on the register allocation policy of the compiler used and on the number of registers available for assignment. For our measurements we use the Portable C Compiler [JOHN79] which is the standard C compiler for UNIX. This compiler selects for allocation in registers only from the variables that have been explicitly defined by the programmer as register variables. Although this does not seem to be a good policy since it results in a large data traffic for the access of variables not allocated in registers [HUGU85], we use it because it is widely available and because we expect the ratio between the traffic for alternative saving/restoring schemes to be relatively independent of the allocation policy.

The number of registers available for assignment is not very critical for the conventional register saving/restoring policies, since the number of register variables per function is quite small [HUGU85]. However, for some of the new policies proposed more registers result in a smaller traffic. To illustrate this point we used six registers (which is the number used by the Portable C Compiler on the VAX-11) and eight registers (this was achieved by a modification of the compiler and of some library functions).

2. Register Saving/Restoring Approaches

We now describe the alternative approaches for register saving and restoring and indicate the support that they require from the architecture and from the compiler. For clarity of exposition we leave the description of the detailed algorithms for Appendix I. Since the objective is to reduce the memory traffic, the approaches are presented in order from the more conventional ones to those that we expect will produce the least traffic. A summary of the schemes is shown in Table 1 and the corresponding measurements are presented in Table 2 and Figure 1.

Policies A and B

Policies A and B are the conventional ones. **Policy A** saves all the registers defined* in the caller when a function is called, and restores the saved registers when the function returns. The architecture can support this policy by including in the call instruction a mask that specifies the registers to be saved/restored.

In **Policy B**, the registers defined in the callee are saved when it is called, and these saved registers are restored upon return. The registers to be saved can be specified in a mask located at the function entry point. Since the return instruction must know which registers to restore, the same mask can be duplicated on each return instruction. The alternative approach of saving the register mask upon function entry, is discarded because it generates two extra memory references for each function call.

McKusick [MCKU84] has evaluated the size of the generated program for both policies when explicit instructions are provided to perform the register saving/restoring. The programs measured were the whole set of UNIX utilities. Since instructions to save/restore the registers must be specified in every call instruction for Policy A and only once (at the entry point) for Policy B, the programs compiled with Policy A resulted 8% bigger than with Policy B. This increase in program size would not be as large if a mask is

*where **defined** means appearing in the program in contrast with used during execution.

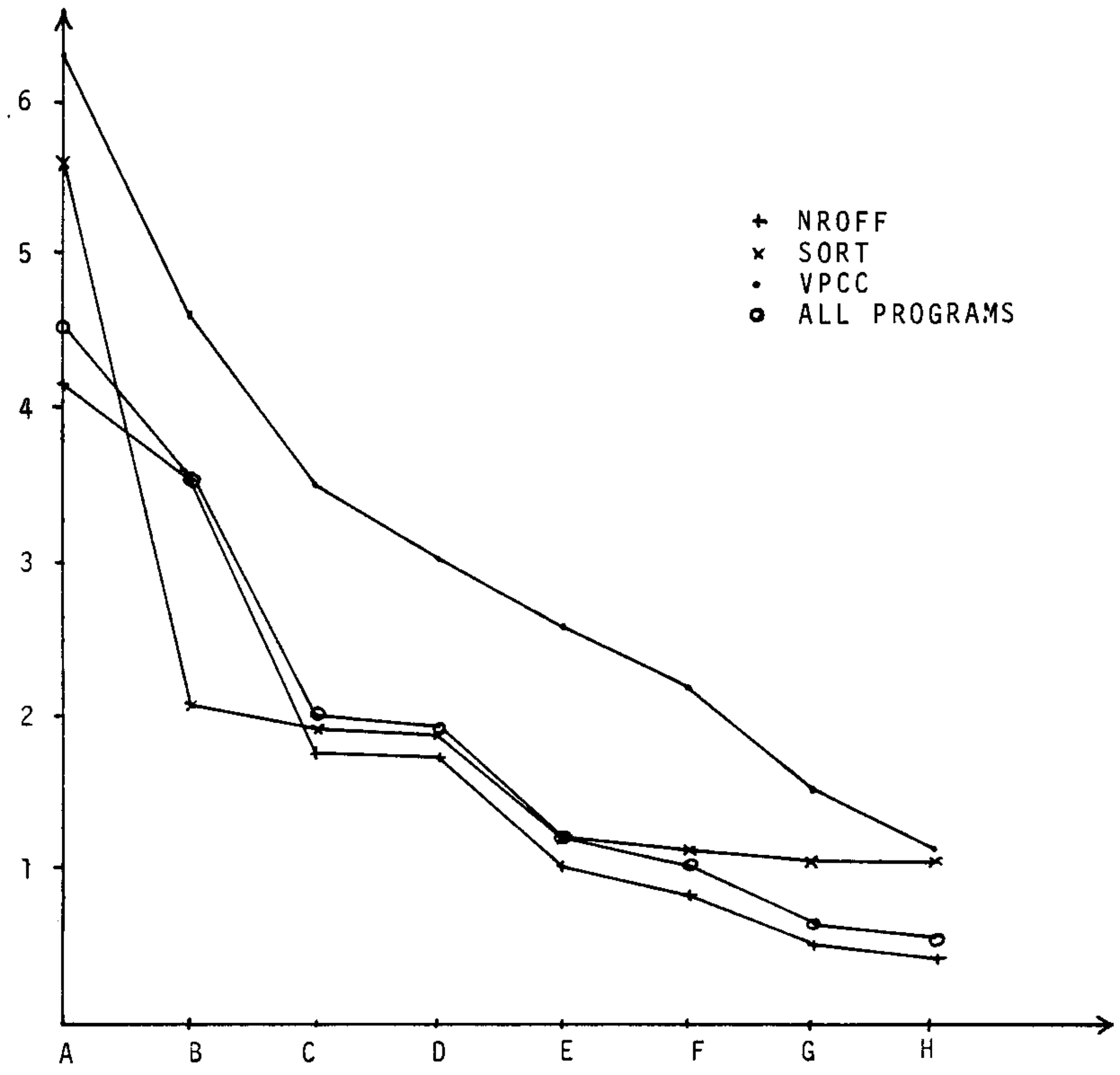

Figure 1. Traffic of Register Saving/Restoring (for 6 registers)

Table 1. Register Saving/Restoring Policies

Save registers	Restore saved registers: on return	Restore saved registers: on return if *already used* by caller	Restore saved registers: when *first read* by caller
on call if *defined* in caller	A	-	-
on call if *defined* in callee	B	-	-
on call if *defined* in callee and *used* in exterior levels	C	-	-
when *used* by callee if *used* in exterior levels	D	-	-
on call if *defined* for callee and *used* in exterior levels and *not saved* yet	-	E	G
when *used* by callee if *used* in exterior levels and *not saved* yet	-	F	H

used, as proposed above. In this case, the instruction memory traffic should be similar for both policies.

From Table 2, we see that, for the composite of all three programs measured, the data memory traffic caused by register saving/restoring is of 3.57 accesses per function call for Policy B. This value is taken as a reference since this policy is the most frequently used. The corresponding traffic for policy A is 27% larger and this policy is consistently worse for all three programs.

In spite of the extra memory traffic generated by Policy A, some compiler writers prefer to use it rather than B [MCKU84]. The reason is that with Policy A the compiler can calculate the cost of allocating a simple local variable to a register because it can estimate how many times it is going to be saved. This information is not known with Policy B and, thus, variables are allocated to registers without considering the cost of having to save/restore them. Moreover, with Policy A the compiler could generate different masks for each function call depending on the registers alive at each point.

It is also worth noticing that due to the nature of these saving/restoring policies there is no reduction in saving/restoring traffic when the number of registers is increased from 6 to 8. We will see that there is some reduction in the other policies.

Policies C and D

The large register saving/restoring traffic for policies A and B is due to the fact that all registers defined for a function are saved, irrespective of their use. That is, a register is saved even if it has not been used previously.

Table 2. Data Memory Traffic Caused by Register Saving/Restoring

Policy	no. reg.	NROFF	SORT	VPCC	ALL PROG
A	6	1.16	2.72	1.37	1.27
	8	1.16	2.72	1.46	1.29
B	6	1.0 (3.59)	1.0 (2.06)	1.0 (4.59)	1.0 (3.57)
	8	1.0	1.0	1.03	1.01
C	6	0.50	0.92	0.76	0.56
	8	0.53	0.58	0.72	0.56
D	6	0.49	0.91	0.66	0.54
	8	0.48	0.57	0.64	0.51
E	6	0.28	0.58	0.56	0.34
	8	0.24	0.21	0.51	0.28
F	6	0.22	0.57	0.47	0.28
	8	0.19	0.19	0.41	0.22
G	6	0.13	0.50	0.32	0.18
	8	0.10	0.16	0.30	0.14
H	6	0.11	0.50	0.29	0.16
	8	0.09	0.15	0.27	0.12
no. of functions		226	21	252	499
no. of calls		1250000	140567	194741	1585308

To reduce the traffic, in policies C and D a register is saved only if it has been used in exterior levels. The management of these policies requires a dynamic mask (TBS = To Be Saved) which specifies the registers that have been used in those levels. This mask has a bit associated to each register; this bit is set when the register is written.

In **Policy C** a register is saved during a call if it is defined for the callee and has been used by exterior levels. The registers saved are restored on return. This dynamic register saving/restoring was proposed for LISP [STEE80].

The architectural support for this policy can consist of the dynamic mask TBS and a static mask per function. During a call, the registers saved correspond to the intersection of both masks, TBS is saved and the bits corresponding to the saved registers are cleared. When a register is written the bit of the mask is set, and during return TBS is restored and the intersection of both masks determines which registers to restore. Note that this mask saving/restoring might increase the data traffic.

To make this policy effective it is necessary to use a compiler that assigns different registers to different functions in order to reduce the intersection. The ideal would be a dynamic assignment that, upon entry to a level, assigns new registers that have not been used by previous levels. This would be very close to what is done in multiple-window register files (with the difference that, to reduce the cost of dynamic mapping, in the multiple-window case registers are saved according to the number defined per function and not to the number used). Here we want to explore the alternative of static mapping and dynamic saving depending on use. For this we use the straightforward scheme of assigning registers in a round-robin fashion as functions are compiled. We modified the Portable C Compiler to generate code using this round-robin register assignment. This modified compiler is used for policies C to H.

Table 2 shows the traffic for register saving/restoring for Policy C. We observe a 44% reduction with respect to Policy B.

In Policy C registers are saved at function entry, whether the register is going to be used or not. To reduce the traffic further, **Policy D** saves the registers when they are going to be written instead of saving them upon function entry.

Two dynamic masks are required in this case: TBS, as before, and CU to indicate the register usage in the current function. This last mask is required for the restoration. Both masks have to be saved/restored.

The architectural support required for this policy is used for manipulating both masks and detecting if a register has to be saved when it is going to be written. During the call, both masks are saved, the TBS is ored with the CU, and the CU is set to zeroes. When a register is written we check both masks: if the register has been used in an exterior level but not in the current function, then it is saved before being written. On return the intersection of both masks determines the registers to restore and both masks are restored.

Since in this case there is no static mask, it is necessary to have some other way of differentiating between registers that have to be preserved across function calls and temporary registers. Two alternatives exist for this: the separation can be fixed, or it can be specified by a global mask. The second solution allows more flexibility because each compiler is allowed to have its own partition of the register set.

Table 2 shows the traffic for Policy D. As we can see, the reduction with respect to Policy C is not significant. This can be due to the following causes: (i) programmers tend to initialize local variables when they are defined (i.e., upon function entry), and (ii) the compiler also initializes the register parameters upon function entry. To improve this policy, an optimizing compiler could possibly delay the initialization until the register is going to be used.

In contrast with policies A and B, we see a reduction in traffic when the number of registers is increased since the probability of using the same register in different levels is reduced.

Policy E

Better use of the registers can be made and the traffic reduced since in policies C and D there is still unnecessary saving and restoring. Consider for example the situation of Figure 2(a); in policy D register R1 is saved and restored in each of the functions Z1, Z2, and Z3 even if it is not used by Y. It is clear that it is sufficient to save it in Z1 and restore it when returning to X. This is done in **Policy E**. Figure 2(b) shows the case in which the function Z is inside a loop. In this case, the previous policies are going to save the register R1 on each call to Z from Y—even though Y has not used the register at all—and to restore it upon return. In contrast, Policy E saves it only once and restores it when returning to X.

To determine when a register has to be saved the TBS mask is used: when a register is written the corresponding bit is set to 1 and when the function returns it is reset. A register is saved if its bit in both the TBS mask and the static mask for that function are 1 (same as in Policy C).

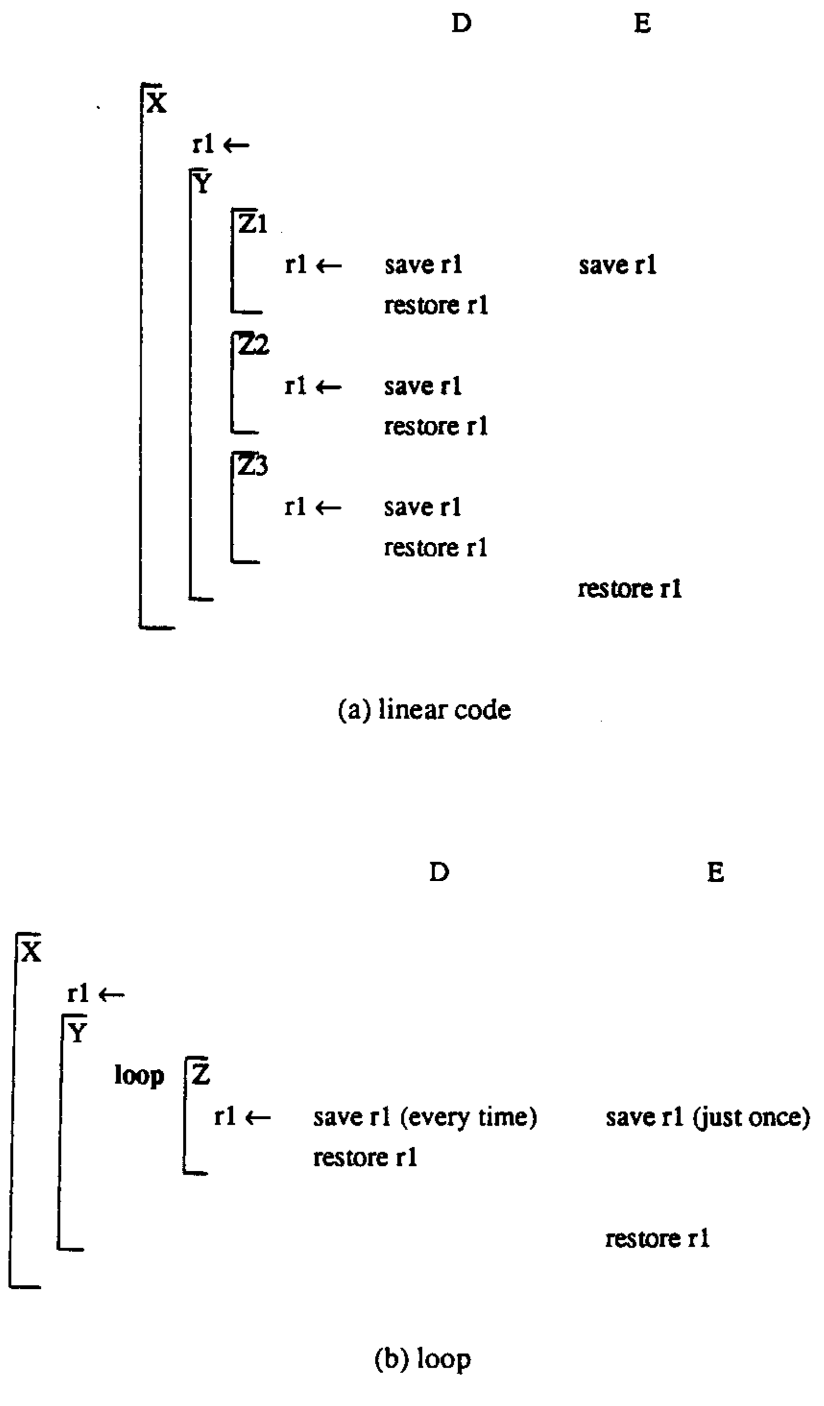

Figure 2. Examples of Policies D and E

To determine when a register is to be restored, a current usage (CU) mask is needed; when returning, if the CU of the caller has the bit set and the TBS mask indicates that it has been saved, the register is restored. To be able to do this restoring it is necessary to save the register in the activation record of the outer function which last used it (X in the example). To do this, the architecture has associated with each register a **pointer** (RP) which is loaded with the current frame pointer each time a register is written. This pointer is used to determine the frame in which the register has to be saved (see Figure 3).

This use of pointers makes the register set operate in a fashion which is similar to a cache. However, the knowledge of the nested nature of the function calls/returns allows the use of the dynamic masks to significantly reduce the traffic which would be obtained in a standard cache implementation.

Only one mask has to be saved/restored: the CU mask.

Table 2 shows the traffic for this policy. For the composite of all three programs the reduction with respect to B is 66% for 6 registers and 72% for 8 registers.

Policies F, G, and H

The traffic produced by Policy E can still be reduced by the following mechanisms:

a) Save a register when it is used (written) instead of at function entry (as in Policy D). This is **Policy F**. This policy uses two dynamic masks in a similar fashion as in Policy D. However, only one mask (CU) needs to be saved/restored.

b) Restore a register when it is needed (read) instead of at function return. This is done in **Policy G**. Note that in this case the restoring has to be done during execution of the instruction; the implementation of this is considered in the next section. This policy uses the static mask and the dynamic TBS mask as Policy C, but the TBS mask does not need to be saved/restored across function calls.

c) A combination of a) and b). This is **Policy H**. This policy uses both dynamic masks and CU must be saved/restored across function calls to know which registers have been used by the current function. Observe that this information is given by the static mask for Policy G which is available in both the call and the return instructions.

The saving/restoring traffic produced when using these policies is shown in Table 2. The minimum traffic occurs with policy H; the reduction with respect to Policy B is of 84% for 6 registers and of 88% for 8 registers. However, this policy has the drawback that one mask has to be saved and restored, which might increase the overall traffic. Because of this, we select for implementation Policy G which has a very similar traffic and requires no mask saving/restoring.

Figure 4 is an example that illustrates the eight policies described.

3. Implementation of the Selected Register Saving/Restoring Policy

In the previous section we selected Policy G as a candidate for implementation because of the small data traffic and because it does not require the saving/restoring of masks. Here we sketch a possible implementation of this scheme to determine its feasability. Of course, alternative implementations are possible and the most suitable one would depend on many design requirements and constraints. The implementation is based on the algorithm given in Appendix I.

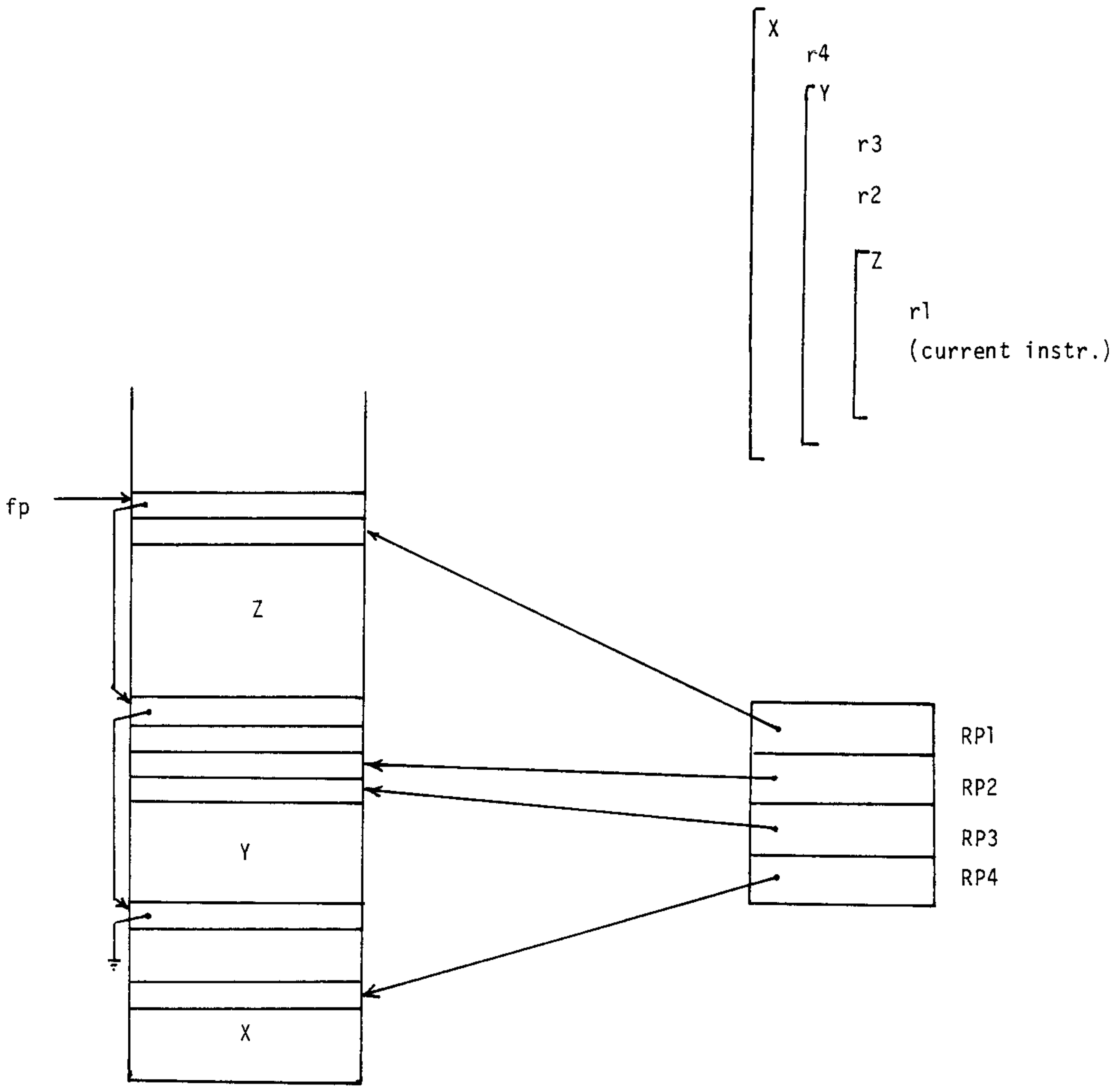

Figure 3. Register Pointers

	A	B	C	D	E	F	G	H
M_x = {r1, r2, r3}		S(1,2,3)						
r1 ←								
r2 ←								
M_y = {r3, r4, r5}	S(1,2,3)	S(3,4,5)						
r3 ←								
M_z = {r3}	S(3,4,5)	S(3)	S(3)		S(3)		S(3)	
r3 ←				S(3)		S(3)		S(3)
	R(3,4,5)	R(3)	R(3)	R(3)	R(3)	R(3)		
r4 ← r3							R(3)	R(3)
M_u = {r1}	S(3,4,5)	S(1)	S(1)		S(1)		S(1)	
r1 ←				S(1)		S(1)		S(1)
	R(3,4,5)	R(1)	R(1)	R(1)				
M_v = {r1, r2, r3}	S(3,4,5)	S(1,2,3)	S(1,2,3)		S(2,3)		S(2,3)	
r1 ←				S(1)				
r2 ←				S(2)		S(2)	S(2)	
	R(3,4,5)	R(1,2,3)	R(1,2,3)	R(1,2)	R(3)			
M_w = {r2}	S(3,4,5)	S(2)		S(2)				
	R(3,4,5)	R(2)	R(2)					
	R(1,2,3)	R(3,4,5)			R(1,2)	R(1,2)		
G ← R1							R(1)	R(1)
		R(1,2,3)						

Figure 4. Example of Register Saving/Restoring Caused by Different Policies

Figure 5 shows the data section of a processor. It has a standard three-bus architecture (which could be changed to one with two buses without significant modifications). Enclosed in the circle we indicate those components that are added due to the register saving/restoring policy. These are:

1) A set of registers RP to contain the pointers. One pointer is required for each register that has to be preserved across function calls. Since these registers are addressed together with the main registers, it is possible to share one of the address decoders amc ng both sets.

2) An adder to compute the address of the memory location where the particular register is saved in the activation record. This address is obtained as the sum of the frame pointer (stored in the specialized register FP) and the register number. In an alternative implementation, this adder could be eliminated and the ALU used for the addition. However, this would require the connection of the RP file to the main data buses which could increase the basic cycle time.

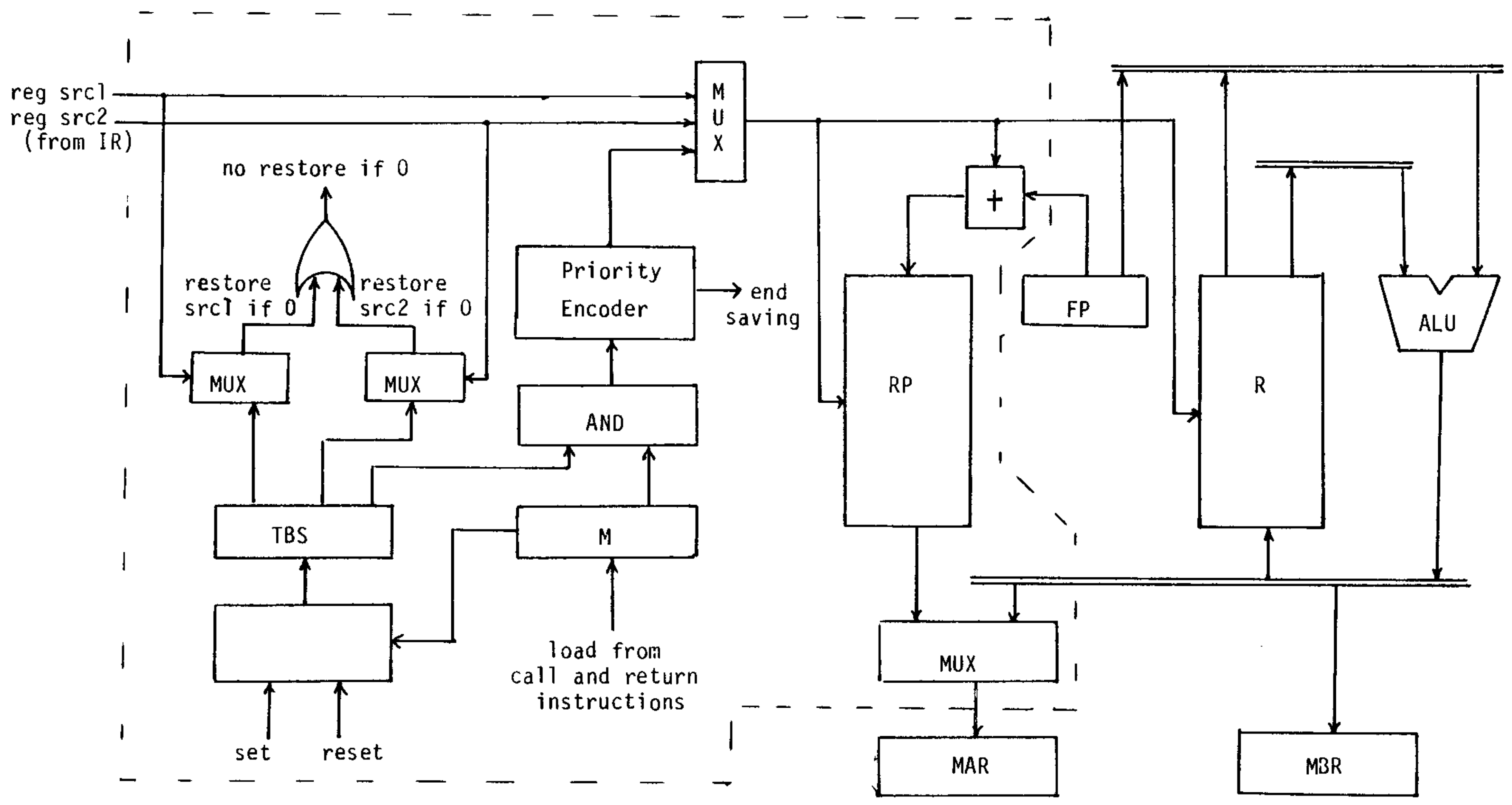

Figure 5. Implementation of Policy G

3) A multiplexer to transfer the pointer to the memory address register. This multiplexer could be eliminated if the RP file is connected to the result bus.

4) Two registers to store the masks.

5) A priority encoder of the AND of both masks to determine the next register to save.

6) Logic to determine whether to restore 0, 1, or 2 registers. This consists of two multiplexers and one OR gate.

7) Logic to set and reset the TBS. A specific bit can be set or reset and several bits can be reset (depending on the mask M).

With this data section the operations related to saving/restoring would be as follows:

i) During a **call** the priority encoder generates, in sequence, the register numbers of those that have to be saved. For each of them, the corresponding pointer is transferred to the memory address register and the values to the memory data register. A memory write is performed and the corresponding bit of the TBS cleared. The clearing of this bit makes the priority encoder produce the next register number.

ii) During a **write** to a register, the corresponding pointer is stored in RP and the corresponding bit of TBS is set.

iii) For **restoring** the fields of the instruction specifying the registers of the operands are used together with the corresponding bits of the TBS to determine if registers have to be restored. If *no-restore = 1*, the operation is executed in a normal manner. If *no-restore = 0*, it is necessary to restore one or two registers. This restoration is done as normal register loads from memory, using as addresses the frame pointer plus the corresponding register numbers. These addresses are also stored in the pointer register file. For each register restored the corresponding bit of TBS is set. After the registers are restored the instruction is executed in the normal way.

iv) During **return** it is necessary to reset the bits of TBS corresponding to the 1's of mask M.

The implementation described indicates that the additional hardware required for this saving/restoring policy consists mainly of a register set for the pointers and of an adder (which could be eliminated as indicated before). The cost, in additional modules or circuit area in a VLSI implementation, can be justified by the reduction in execution time provided by the policy. The basic processor cycle is not increased by these additions since they are decoupled from the main datapaths and the operations that have to be done during normal register reads and writes are performed overlapped with the normal cycle.

Since most modern processors use pipelining to increase the throughput, it is important to evaluate the impact of this saving/restoring policy and its implementation on the performance of such a processor. The following considerations apply to this:

a) Since a dynamic mask is used during call to determine which registers to save, it is necessary to wait until this mask is set (by previous instructions) before executing the call. However, this does not introduce additional constraints in most processors because they would anyhow wait before executing a call (because of the change in environment).

b) During a return bits of TBS are reset. This can be done without waiting for the previous instructions to finish, if the eventual setting by them of the TBS is inhibited. However, similar to a call, usually a return would anyhow be a waiting point.

c) During a register write no additional requirements are imposed.

d) Before a register read it is necessary to finish any previous restoring of that register. The control of this dependency can use the same flag required for the control of the read-after-write condition.

From the previous discussion we conclude that the implementation of the proposed saving/restoring policy does not impose significant limitations to the pipelined implementation of the processor.

4. Conclusions

We have described and evaluated several register saving and restoring policies that provide a reduction of memory traffic with respect to the conventional ones used by most processors. We also show that these policies could benefit from a larger set of registers, which is not the case for the conventional ones if the allocation policy of the Portable C Compiler is used.

From the policies proposed we have selected Policy G for implementation because, for the measured programs, it produces a reduction of 86% in the traffic and does not require the saving/restoring of any masks. We have sketched a possible implementation and concluded that the amount of additional hardware required is sufficiently small to justify the inclusion because of the reduction in execution time that would result. We also concluded that the added hardware would not increase the basic cycle of the processor and that the policy does not pose significant limitations on a pipelined implementation of the processor.

Appendix I

This appendix describes the actions performed for the eight register saving and restoring policies discussed in Section 2. The algorithms are specified using the following notation:

R	Set of general-purpose registers
R^s	Subset of registers preserved across function calls
RP	Register Pointers. Set of specialized registers associated to *R* (to indicate the location of the register in the activation record where *R* was used)
X[i]	element *i* of the set *X*, where *X* is *R*, R^s, or *RP*
M	register mask given at the function entry point (to indicate registers defined by the function)
CU	Current Usage mask (to indicate registers used by the current function)
TBS	To Be Saved mask (to indicate registers used by the exterior levels)*
m<i>	bit *i* in the mask *m*, where *m* is *M*, *CU*, or *TBS*

Policy A. Save the registers defined for the caller and restore them upon return. The operations to perform are:

on call	save registers given by the register mask of the call instruction
on return	restore registers given by the register mask of the call instruction

Policy B. Save the registers defined for the callee and restore them upon return. The operations to perform are:

on call	save registers given by the register mask at function entry
on return	restore registers given by the register mask in the return instruction

Policy C. Save the registers defined for the callee if they have already been used; restore them upon return. TBS and M indicate which registers have to be saved and restored. The operations to perform are:

```
on write (R[i])    TBS<i> ← 1;
on call            save TBS;
                   for all i do if (TBS<i> = 1 and M<i> = 1) then
                           save R[i];
                           TBS<i> ← 0;
on return          restore TBS;
                   for all i do if (TBS<i> = 1 and M<i> = 1) then restore R[i];
```

*For some policies TBS indicates both the registers used by the current function and the registers used by the exterior levels.

Policy D. Save a register when it is first written (by the callee) if it has already been used; restore it upon return. This policy uses both masks (TBS and CU) to indicate which registers need to be saved and restored. The operations to perform are:

```
on write ($R^S[i]$)   if (TBS<i> = 1 and CU<i> = 0) then save R[i];
                      CU<i> ← 1;
on call               save TBS;
                      save CU;
                      TBS ← TBS or CU;
                      CU ← 0
on return             for all i  do if (TBS<i> = 1 and CU<i> = 1) then restore R[i];
                      restore TBS;
                      restore CU;
```

Policy E. Save the registers defined by the callee that have already been used and have not been saved yet; restore registers saved on return if they have been used by the caller. This policy uses all three masks (TBS, CU, and M), but only the CU mask has to be saved and restored across function calls. The operations to perform are:

```
on write (R[i])       TBS<i> ← CU<i> ← 1;
                      RP[i] ← current frame pointer + displacement;
on call               save CU;
                      for all i  do if (TBS<i> = 1 and M<i> = 1) then
                              save R[i] in the memory location given by RP[i];
                              TBS<i> ← 0;
                      CU ← 0;
on return             for all i  do if (M<i> = 1) then TBS<i> ← 0;
                      restore CU;
                      for all i  do if (TBS<i> = 0 and CU<i> = 1) then
                              restore R[i];
                              RP[i] ← caller's frame pointer + displacement;
                              TBS<i> ← 1;
```

Policy F. Save a register when it is first written (by the callee) if it has already been used and has not been saved yet; restore registers saved on return if they have been used by the caller. This policy uses both the TBS and the CU masks, but only the CU mask needs to be saved and restored across function calls. The operations to perform are:

```
on write ($R^S[i]$)   if (TBS<i> = 1 and CU<i> = 0) then
                              save R[i] in the memory location given by RP[i];
                              RP[i] ← current frame pointer + displacement;
                      TBS<i> ← CU<i> ← 1;
on call               save CU;
                      CU ← 0;
on return             for all i  do if (CU<i> = 1) then TBS<i> ← 0;
                      restore CU;
```

```
                    for all i  do if (TBS<i> = 0 and CU<i> = 1) then
                              restore R[i];
                              RP[i] ← caller's frame pointer + displacement;
                              TBS<i> ← 1;
```

Policy G. Save the registers defined by the callee that have already been used and have not been saved yet; restore registers saved when they are first read by the caller. This policy uses the M and TBS masks, but no mask needs to be saved and restored across function calls. The operations to perform are:

```
on read (R[i])      if (TBS<i> = 0) then
                              restore R[i];
                              RP[i] ← current frame pointer + displacement;
                              TBS<i> ← 1;
on write (R[i])     TBS<i> ← 1;
                    RP[i] ← current frame pointer + displacement;
on call             for all i  do if (TBS<i> = 1 and M<i> = 1) then
                              save R[i] in the memory location given by RP[i];
                              TBS<i> ← 0;
on return           for all i  do if (M<i> = 1) then TBS<i> ← 0;
```

Policy H Save a register when it is first written (by the callee) if it has already been used and has not been saved yet; restore registers saved when they are first read by the caller. This policy also uses both the TBS and the CU masks, but only the CU mask needs to be saved and restored across function calls. The operations to perform are:

```
on read($R^S$[i])     if (TBS<i> = 0 and CU<i> = 1) then
                              restore R[i];
                              RP[i] ← current frame pointer + displacement;
                              TBS<i> ← 1;
on write ($R^S$[i])   if (TBS<i> = 1 and CU<i> = 0) then
                              save R[i] in the memory location given by RP[i];
                              RP[i] ← current frame pointer + displacement;
                    TBS<i> ← CU<i> ← 1;
on call             save CU;
                    CU ← 0;
on return           for all i  do if (CU<i> = 1) then TBS<i> ← 0
                    restore CU;
```

Appendix II

This appendix contains the data memory traffic caused by register saving and restoring (DMT/SR) for the small benchmark programs. Twelve programs have been measured: five (E, F, H, I, K) were developed at Carnegie-Mellon University to evaluate different computer architectures [GRAP81], and they were coded in C by Piepho [PIEP81]; the Ackermann's function (acker) [WICH76]; the C version of the Dhrystone benchmark [WEIC84]; two versions of the Puzzle benchmark, one using pointers (puzzptr) and the other using subindexes (puzzsub) [BEEL84]; the Quicksort benchmark (qsort); the Sieve of Eratothenes benchmark (sieve) [GILB81]; and the Towers of Hanoi benchmark (towers).

Table 3 gives the register saving/restoring traffic for the standard version of the programs. Since some of these programs do not use register variables at all*, new versions were developed to force register allocation by the Portable C Compiler. Table 4 gives the traffic for the new versions.

*Register saving and restoring traffic is generated by the library functions.

Table 3. DMT/SR Caused by the Benchmark Programs

program	no. reg.	no. ftn	no. calls	policy A	B	C	D	E	F	G	H
E	6	2	10	2.00	2.10	0.20	0.20	0.20	0.20	0.30	0.10
	8							0.30		0.50	
F	6	2	12	0.00	6.08	0.00	0.00	0.00	0.00	0.00	0.00
	8										
H	6	2	8	0.00	0.13	0.00	0.00	0.00	0.00	0.00	0.00
	8										
I	6	2	4	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00
	8										
K	6	2	4	0.00	3.25	0.00	0.00	0.00	0.00	0.00	0.00
	8				3.75						
acker	6	2	172236	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	8										
dhrystone	6	12	170055	0.00	0.59	0.00	0.00	0.00	0.00	0.00	0.00
	8										
puzzptr	6	5	21381	5.99	9.80	8.58	9.71	4.59	4.58	3.64	3.64
	8							4.21	4.21	3.43	3.43
puzzsub	6	5	21381	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
	8										
qsort	6	6	42621	10.00	2.25	2.25	2.06	1.74	1.55	1.63	1.51
	8					2.10	1.91	1.79	1.60	1.51	1.39
sieve	6	1	9	2.22	2.33	0.22	0.22	0.22	0.22	0.11	0.11
	8							0.33		0.33	
towers	6	2	524290	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	8										
TOTALS	6	43	952011	0.58	0.43	0.29	0.31	0.18	0.17	0.16	0.15
	8						0.30	0.17		0.14	0.14

Table 4. DMT/SR Caused by the Benchmark Programs (with registers)

program	no. reg.	no. ftn	no. calls	policy							
				A	B	C	D	E	F	G	H
Ereg	6	2	10	4.00	4.30	2.00	2.20	2.00	1.80	0.80	0.60
	8					1.60	1.80	1.70	1.40	1.00	0.50
Freg	6	2	12	3.00	7.92	1.33	1.33	1.33	1.33	0.67	0.67
	8					0.00	0.00	0.00	0.00	0.00	0.00
Hreg	6	2	8	1.25	5.38	0.00	0.00	0.00	0.00	0.00	0.00
	8					0.00	0.00	0.00	0.00	0.00	0.00
Ireg	6	2	4	2.00	5.25	2.00	2.00	2.00	2.00	1.25	1.00
	8					1.00	1.00	1.00	1.00	0.75	0.50
Kreg	6	2	4	1.00	4.25	1.00	1.00	1.00	1.00	0.50	0.50
	8					0.00	0.00	0.00	0.00	0.00	0.00
ackerreg	6	2	172236	4.00	4.00	4.00	4.00	4.00	4.00	2.00	2.00
	8										
dhrystonereg	6	12	170055	5.29	4.12	2.94	2.94	2.47	2.47	1.00	1.00
	8					1.76	1.76	1.65	1.65	0.59	0.59
puzzptr	6	5	21381	5.99	9.80	8.58	9.71	4.59	4.58	3.64	3.64
	8							4.21	4.21	3.43	3.43
puzzsubreg	6	5	21381	5.99	6.00	4.28	4.28	1.04	0.94	0.61	0.52
	8							3.53	3.44	3.41	3.32
qsort	6	6	42621	10.00	2.25	2.25	2.06	1.74	1.55	1.63	1.51
	8					2.10	1.91	1.79	1.60	1.51	1.39
sievereg	6	1	9	3.33	3.44	0.22	0.22	0.22	0.22	0.11	0.11
	8							0.33		0.33	
towersreg	6	2	524290	8.00	8.00	8.00	8.00	8.00	8.00	6.00	6.00
	8										
TOTALS	6	43	952011	6.79	6.32	6.04	6.06	5.77	5.76	4.01	4.01
	8					5.83	5.84	5.68	5.67	3.99	3.98

References

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

[FURH85] B. Furht, ''RISC Architectures with Multiple Overlapping Windows,'' *Compsac 85* (October 1985).

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

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

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

[HITC85] C. Y. Hitchcock and H. M. Brinkley Sprunt, ''Analyzing Multiple Register Sets,'' *The 12th Annual International Symposium on Computer Architecture*, pp. 55-63 (June 1985). Published in *Computer Architecture News* **13**(3).

[HUGU85] M. Huguet, ''A C-Oriented Register Set Design,'' Master's Thesis, University of California, Los Angeles, CA 90024, (June 1985). Also published as Technical Report No. CSD-850019.

[HUGU85a]M. Huguet and T. Lang, ''A C-Oriented Register Set Design,'' *Proceedings of the 29th Symposium on Mini and Microcomputers and their Applications*, pp. 182-9 (June 1985).

[HUGU85b] M. Huguet and T. Lang, ''A Reduced Register File for RISC Architectures,'' *Computer Architecture News* **13**(4), pp. 22-31 (September 1985).

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

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

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

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

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

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

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

[STEE80] G. L. Steele Jr. and G. J. Sussman, "The Dream of a Lifetime: A Lazy Variable Extent Mechanism," *Conference Record of the 1980 LISP Conference*, pp. 163-72 (August 1980).

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

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