SIMULATION AND PERFORMANCE EVALUATION OF THE RISC ARCHITECTURE

by

Y. Tamir

Memorandum No. UCB/ERL M81/17

6 April 1981

SIMULATION AND PERFORMANCE EVALUATION
OF THE RISC ARCHITECTURE

by

Yuval Tamir

Memorandum No. UCB/ERL M81/17

6 April 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

UNIVERSITY OF CALIFORNIA

College of Engineering

Department of Electrical Engineering

and Computer Sciences

Computer Science Division

Simulation and Performance Evaluation of the RISC Architecture

by

Yuval Tamir

Abstract

The Reduced Instruction Set Computer (RISC) is an attempt to achieve high performance without resorting to complex instructions and pipelining schemes. The success or failure of this novel architecture can only be determined by its performance in executing "real" programs. The simulator of the RISC architecture, described in this paper, makes possible detailed studies of the performance of RISC even though the hardware implementation of RISC is not complete.

March 1981

1. Introduction

Computer architecture is not an exact science. There is no generally accepted definition of what constitutes a "good" architecture or of what is the best measure of the "quality" of the architecture. Some traditional measures of the "quality" of the architecture are: performance - speed (for a given class of applications), code size, "ease of programming" and operating system support.

When a new architecture is proposed, its novel features should be evaluated using objective criteria. Some of the information (such as cache statistics, number of memory references and number of machine instructions executed for a given task) required to evaluate the new features is often obtainable only by executing programs on hardware implementing the architecture *or* on a simulator for the architecture.

The RISC architecture is an attempt to achieve high performance by a judicious choice of a simple instruction set and a special mechanism for supporting subroutine calls[5]. A C compiler[1], peephole optimizer[2], and assembler[6] for RISC have been developed. This paper describes a simulator for the RISC architecture which is capable of simulating the execution of substantial programs and producing instruction and data address traces. The execution of a few programs is simulated and a *trace analyzer*, developed as part of this project, is used to obtain dynamic opcode frequencies. These results are used for a preliminary evaluation of the novel features of the RISC architecture.

2. The RISC Architecture

RISC is a 32 bit, register oriented machine. Instructions are all the same length (32 bits). Only load and store instructions can reference memory and the only addressing mode for loads and stores is indexed addressing. Details of the architecture, including the instruction set and its justification, can be found in a paper by Patterson and Séquin[5]. The three novel features of this architecture are: a simple instruction set with a single instruction size (32 bits), a special mechanism for subroutine calls, and a "delayed branch" scheme that makes possible simple instruction prefetch. These three features are evaluated in this paper and are therefore briefly described in this section.

2.1. The Simple Instruction Set

"A general trend in computers today is to increase the complexity of architectures commensurate with the increasing potential of implementation technologies"[5]. This has led to increased design time, increased design errors and inconsistent implementations[4]. The RISC project attempts to show that this increase in complexity doesn't necessarily lead to improved performance and that, in fact, there are performance benefits to using a simple architecture which allows a hardware (as opposed to microprogrammed) implementation of the control unit resulting in fast execution of each instruction.

The RISC instruction set contains only simple instructions: register to register arithmetic, logical and shift operations, *load*'s and *store*'s with a single addressing mode (-indexed addressing of the type register+register or register+displacement), relative *jump*'s and conditional relative *jump*'s, indexed *jump*'s and conditional indexed *jump*'s, *call* and *return*. Due to the fixed size of the instructions only 13 bit immediates can be used. Each time a larger immediate is needed, two extra instructions must be executed to "construct" the full 32 bit constant.

All complex operations such as multiply/divide, context switch, array indexing and bounds checking, *etc.* must be done in software.

2.2. Multiple Register Banks

Investigations of the use of high-level languages show that subroutine calls (with all the related overhead of passing parameters and saving and restoring of registers) are probably the most time consuming operation in typical programs[3]. In order to avoid saving and restoring of registers, multiple banks of registers are kept in the CPU. A new bank of registers is allocated as a result of a subroutine call by simply modifying a pointer. A return from a subroutine simply changes the pointer to the previously used bank so all the registers are restored without *any* memory references. In order to facilitate the passing of parameters, the register banks *overlap* so that the "output" registers of the register bank of the calling routine are the same storage elements as the "input" registers of the register bank of the called routine. Thus, scalar parameters can be passed without any memory references.

Since the number of register banks is limited, there is a need for a mechanism which will handle the case when the subroutine nesting "depth" is more than the number of register banks. This is done by having a conventional stack which is used to store "overflowed" banks.

In the architecture being investigated, instructions may access 32 registers. Of these, 10 registers are fixed (i.e. all subroutines refer to the same storage elements), and 22 registers are in the register bank. Six registers are "input" registers (for parameters from the calling routine), ten are "local" registers (for local scalars), and six are "output" registers. There are eight register banks, one of which is always empty (for reasons beyond the scope of this paper). When an attempt is made to allocate the eighth bank (overflow), some of the banks are moved to memory. When a *return* is executed and there are no allocated banks (underflow), a few register banks are loaded from memory.

2.3. Delayed Jumps

The performance of RISC is increased by prefetching the next instruction while executing the current instruction. In most existing computers, elaborate techniques are required to prefetch the appropriate instruction after a branch (where "branch" includes *jump*'s, *call*'s and *return*'s). The RISC solution to this problem is to always execute the instruction which follows the branch in the code and have the branch take effect only for the instruction *following* the instruction that follows the branch. This scheme is called *delayed jump*.

The compiler[1] always places a *nop* following the *jump*. The peephole optimizer[2] attempts to replace the *nop* with some useful instruction. The main question here is whether the optimizer is successful in a sufficient percentage of the branches so that the added complexity introduced by the *delayed jump* is justified.

3. The Design of a Simulator for the RISC Architecture

The simulator of the RISC architecture was developed in order to be able to investigate the dynamic "performance" of the architecture before a hardware implementation is complete. This simulator, written in C on a VAX running Berkeley UNIX, reads the object file created by the assembler and "executes" the program interpreting each instruction every time it is executed. The register banks as well as the program counter, stack pointer and condition code flags, can be represented by simple data structures due to the fact that the VAX is also a 32 bit machine.

The first task in the design of the simulator was to define the type of programs to be simulated and the level at which they will be simulated. Ideally, the execution of an entire operating system with "typical" user processes should be simulated. This is not practical for two reasons: (1) The ratio of simulation time to simulated time is on the order of 100 to 1000, so simulating the execution of an entire operating system will simply take too much time and too many resources on the host computer (in this case the VAX). (2) An operating system for RISC doesn't exist yet.

Even if the operating system is not simulated, the simulator must still deal with operating system issues: Every useful program uses input/output, many use dynamic memory allocation. The simulator developed handles this problem by translating system calls from the program whose execution is being simulated, to "real" UNIX system calls on the VAX. This speeds up the simulation since the system call is executed on the VAX rather than simulated. In addition the statistics obtained from the simulation will depend only on the program whose execution is being simulated (whose selection is obviously completely up to the user), and not on the way that the system routines are implemented.

The translation of simulated system calls to real system calls is done by having the assembler assume that the entry points of recognized system calls are at addresses where valid code can never reside ("negative" addresses starting −1 are used). When the simulator is directed, by the object code, to execute an instruction at a "negative" address, it assumes that a system routine has been called and determines which system routine has been called by the value of the negative address. Thus from the user's point of view it appears that all system calls are done "by magic", take no (simulated) time and produce no memory references.

Several tables for translating file descriptors and file pointers in the simulated program to "real" file descriptors and file pointers facilitate efficient handling of I/O system calls. An additional table is used for translating memory references of the simulated program to real VAX addresses. These address translations table is the same kind of table used in a virtual memory system. Memory is divided into 1K byte *pages* and for each page there is an entry in the translation table which specifies its real address.

The register banks overflows and underflows, described in the previous section, are also handled "by magic". No extra memory references are executed and no extra instructions will appear in the statistics obtained from the simulation. The main reasons for this "feature" are: (1) The architecture specifies that register banks overflows and underflows cause traps. Unfortunately, the RISC routine for handling traps in general and register overflows and underflows in particular doesn't exist yet. (2) Handling register banks overflows and underflows "by magic" increases simulation speed. It should be noted that since the instructions executed in order to handle the register banks overflows and underflows will always be the same, their effect can easily be added to the statistics by the *trace analyzer* described below.

Clearly, simply simulating the execution of the program is of no interest.

The purpose of the simulation is to obtain dynamic statistics and address traces from the execution of the program. There are two places where statistics can be collected: (1) In the simulator itself. (2) In *trace analyzers* which read the instruction and data address traces produced by the simulator. The advantage of collecting statistics in the simulator itself is that it is faster than producing address traces and analyzing them by special programs. The disadvantage of collecting statistics in the simulator itself is that it clutters the simulator code making it less readable. Furthermore, when new types of statistics are desired, the simulator itself must be modified thereby possibly introducing bugs.

The RISC simulator collects basic dynamic statistics (namely, opcode frequencies) and/or produces instruction and data address traces. The address traces are compressed so that only the minimum number of bytes is stored in the trace file or piped to the trace analyzer. Even though all addresses are 32 bits, most of them are redundant (there aren't many programs whose data segment and text segment sizes are more than 64K bytes). A simple sign extension scheme reduces the number of trace bytes substantially. For instruction addresses, instead of storing the address of the instruction, the offset from the address of the last instruction is stored. If the instruction follows the previously executed instruction in memory, only a control byte is generated. Thus for each instruction which was not reached by a branch and which doesn't access memory, only one byte (the control byte) is generated.

Trace analyzers are easy to develop. As part of this project a trace analyzer which produces opcode frequencies, opcode pair frequencies and register reference counts was developed. This program reads the trace output from the simulator *and* the object file and produces its results by relating instruction addresses to instructions in the object file. In addition, a trace analyzer which produces cache statistics (hit ratios) has also been developed. This program will be used in the future for designing a cache for RISC.

The appendix describes the details of how a given C program can be compiled, optimized and assembled. It also describes how the resulting object file can be used to simulate the execution of the program and obtain dynamic statistics.

4. Unexpected Uses of the Simulator and Measurements of its Performance

The intended use of the simulator was to evaluate the RISC architecture. As it turned out, an equally important use of the simulator was for debugging the C compile[1] and peephole optimizer[2]. Once the simulator was complete and capable of simulating the execution of large programs, it took an additional ten days to debug the compiler and optimizer which were thought to be operational. A few subtle bugs showed up only in relatively complex programs such as the UNIX batch editor *sed*. Such bugs could not be detected without the aid of the simulator. Thus an important use of the simulator is debugging software for the proposed architecture. This ensures that once the hardware implementation of RISC is complete, important software tools (such as the C compiler) will be quickly ready for use since they can be tested and debugged in parallel with the development of the hardware rather than after the hardware is complete.

Another application of the simulator is in obtaining information that will help in making the optimizer generate more efficient code. In some cases, the optimizer replaces *nop*'s that follow *jumps* with the instruction at the destination of the jump. In the case of conditional *jumps* there is sometimes a choice of replacing the *nop* with the instruction at the destination of the *jump* or the instruction that follows the *nop* in memory. The ideal choice is to use the instruction that will be needed more often. In order to make this choice the optimizer has to know whether the jump is more likely to be taken or not (i.e. whether the conditions for the *jump* are usually satisfied of not). Using the instruction address trace, produced by the simulator, and the *trace analyzer*, it is possible to collect statistics that will show, for each type of conditional *jump* and each direction of jump, whether the conditions for the *jump* are usually satisfied or not. It is hoped that, by collecting this information for several benchmark programs, the probable behavior of most programs can be deduced and this information can be used to make the optimizer generate more efficient code.

Since the simulator must interpret each instruction each time it is executed and perform the functions of the single instruction using many instructions on the host computer, it is clear that a large simulation time to simulated time ratio is to be expected. For this simulator, for all the programs whose execution was simulated, this ratio was found to be around 400:1. The simulated time was determined using the dynamic statistics that show the number of instructions of each type that were executed and assuming that each *load* or *store* takes 800 nsecs and all other instructions take 400 nsecs (this assumption is based on the VLSI implementation of RISC being developed in Berkeley). The simulation time was determined using the *time* command and/or the C language profiler. From the profiler it can be determined that most of the simulation time (about 60%) is spent in executing the functions of the instruction and about 20% is spent in translating RISC addresses to "real" addresses on the VAX.

It should be noted that piping the address trace to the opcode trace analyzer (see appendix) significantly increases the simulation time. For all the benchmarks used, the execution time (CPU time) of the trace analyzer was about 60% that of the simulator.

5. Evaluation of the RISC Architecture

As indicated in section 2, the three novel features of the RISC architecture are: a simple instruction set, the multiple register bank scheme, and the delayed jumps. Out of the many possible measures of the effectiveness of these features, this paper will concentrate on performance (i.e. the time it takes to perform a given task), since high performance (with a simple architecture) is the major goal of the RISC architecture. Since the execution time is needed, some assumptions must be made about the execution time of each instruction. As in section 4 (and with the same justification), the assumption is that each *load* or *store* (i.e. each memory access) takes 800 nsecs and all other instructions take 400 nsecs.

Ideally, each one of the novel features of the RISC architecture should be evaluated independently. For example, it would be interesting to evaluate the effectiveness of a simple instruction set **without** the multiple register banks by having a compiler, optimizer, assembler, simulator package which used the simple instruction set but used a conventional stack to store registers and pass parameters in a subroutine call. Unfortunately, the software development effort required for this approach makes it impractical in our research environment. Therefore the effectiveness of each one of the features has to be deduced from data obtained from the execution (simulation) on the full RISC architecture.

The choice of benchmarks is a critical part of every performance study of this kind. The benchmarks are usually chosen to reflect the expected (or measured) workload of the system. In this case, there is no working system to measure. A reasonable future application for RISC is as a personal computer running UNIX. Hence the "expected load" may be chosen by considering "typical" UNIX workloads.

As indicated in section 3, simulating the complete operating system and its workload is not practical. Furthermore, many programs include references to system data structures or to system programs which, in turn, refer to system data structures. Simulating the execution of this kind of programs is also not practical since it requires "simulating" many system data structures and/or having to compile a long "chain" of system routines which call one another. Hence the choice of benchmark programs is rather limited.

The first two benchmarks to be used in this study are based on a program called "puzzle" developed by Forest Baskett. This is essentially a bin-packing program that solves a three-dimensional puzzle. It was chosen mainly because it has been run on many different computers and can therefore be used to compare RISC to the state of the art in commercial computers. There are two versions of this program: The first, called *puzzle*, uses subscripts to access arrays. The second, called *ppuzzle* uses pointers to access arrays.

The next benchmark to be used is the UNIX recursive quicksort program being used to sort 2600 integers, in the range (0,100000) that were generated by UNIX's multiplicative congruential random number generator *rand*. This program will be referred to as *qsort*. *Qsort* was chosen since sorting is a common operation which is important for most compilation and text processing tasks which are heavily used on UNIX.

Possibly the most interesting benchmark is UNIX's batch editor *sed* being used to run a few (admittedly random) commands on the manual for the *make* program (this choice was also quite random). This program was chosen because editing is one of the most important tasks in an interactive system and the type of operations performed by the script used (searching for a pattern and substituting another pattern) are typical of most programs that handle ASCII text as their input.

The final benchmark is a Tower of Hanoi program being used to move

18 disks. It was chosen mainly in order to demonstrate what happens with a program which consists, almost entirely, of subroutine calls. This program will be referred to as *tower*.

The basis for the discussion in this section will be the data in Table 1. All the RISC data was obtained using the RISC simulator and opcode trace analyzer. The VAX data on *puzzle*, *ppuzzle* and *qsort* was obtained by Patterson and Séquin[5] using an instruction trace program. The VAX execution time for *sed* and *tower* were obtained using the *csh* **time** command and the C profiler. (Using the profiler, the time spent in system I/O routines, which is not taken into account in the RISC simulation, was subtracted from the execution time reported by **time** in order to make fair comparisons with RISC). The number of data memory references for *sed* and *tower* for the VAX has not been determined. The programs run on the VAX were all optimized. Both optimized and unoptimized versions of the programs were run on the RISC simulator.

The RISC statistics presented in Table 1 are only a small part of the wealth of information produced by the simulator and trace analyzer. The interpretation of some of the items is not obvious: **ldhi** is an instruction which is only used when a constant which requires more than 13 bits is needed. The "# instructions simulated" is the number of instructions actually simulated and it does not include the handling of register banks overflows/underflows or the execution of any of the routines which are "trapped" by the simulator and executed "by magic" (see section 3). The #overflows+#underflows reported is the number of register banks overflows and underflows under the assumption that whenever there is an overflow, four register banks are stored in memory and whenever there is an underflow, four register banks are loaded from memory.

The "estimated RISC execution time" is obtained by adding an estimate of the time it takes to handle register banks overflows and underflows to the time

Table 1
Summary of Dynamic Statistics

		puzzle	*ppuzzle*	*qsort*	*sed*	*tower*
# jumps (M)	optimized	1.728	1.728	0.228	0.946	0.524
	unoptimized	1.791	1.790	0.388	1.098	1.835
# calls (M)		0.021	0.021	0.055	0.015	0.524
# ldhi (M)	optimized	0	0	0	0.106	0
	unoptimized	0.004	0	0	0.254	0
# nops (M)	optimized	0.017	0.673	0.016	0.200	0.524
	unoptimized	1.812	1.812	0.441	1.113	2.359
# data memory accesses (M)	optimized	1.704	0.953	0.399	1.478	0
	unoptimized	1.704	0.953	0.401	1.478	0
# instructions simulated (M)	optimized	10.115	7.104	1.493	4.635	5.505
	unoptimized	11.196	7.345	2.018	5.802	8.389
Maximum Nesting Depth		20	20	10	6	20
# overflows + # underflows		124	124	64	0	69904
Estimated RISC execution time (secs)	optimized	4.736	3.231	0.932	2.445	6.620
	unoptimized	5.168	3.327	1.143	2.912	7.773
Net VAX execution time (secs)		11.3	4.0	1.8	2.57	14
# data memory accesses on the VAX (M)		5.8	1.4	1.4	-	-

is takes to execute the instructions simulated. In the case of *qsort*, the estimate also includes the time it takes to do 1713 integer multiplies and 1712 integer divisions (The multiply and divide routines are currently trapped by the simulator and handled "by magic" without producing any dynamic statistics). The time to handle register banks overflow (underflow) was estimated by assuming that each time an overflow or underflow occurs 30 instructions are needed for "bookkeeping" (interrupt/trap handling) in addition to one *store* (*load*) for each register whose register bank is being stored in memory (loaded from memory). For multiplies and divides, it is assumed that on the average, each multiply takes 50 instructions and each divide takes 200 instructions. It should be noted that for the first four benchmarks (i.e. all the benchmarks except *tower*), the estimated time for handling register banks overflows and underflows and multiplies and divides is much less than the time is takes to execute the simulated instructions. Hence, gross errors in the assumed number of instructions needed to handle overflows/underflows multiplies/divides, would not significantly change the total execution time of the program.

The results show that for all the benchmarks, optimized programs on RISC ran as fast as or faster than the same programs on the VAX 11/780 (a successful modern minicomputer). The question is which of the three novel features of the RISC architecture is (are) mainly responsible for this impressive performance.

The effectiveness of the simple instruction set is evaluated by comparing the execution time to that of an existing successful machine which has a complex instruction set, namely the VAX 11/780. First, it should be noticed that for all the benchmarks, except *sed*, even the unoptimized programs ran faster on RISC than on the VAX. Unoptimized programs take advantage of the delayed jump scheme only after *call* instructions (where the instruction that follows the call saves the stack pointer). If the delayed jump scheme was not used at all, an upper bound on the total execution time is the time it takes to execute the unoptimized programs plus the time it takes to execute *n add* instructions where *n* is the number of *call*'s executed and *add* is the instruction which is used to save the stack pointer. From the data, for all the benchmarks, except *sed*, the upper bound on the execution time without delayed jumps still shows RISC running faster than the VAX. Hence the delayed jump scheme is not the key to RISC's success.

In order to evaluate the overlapping register banks scheme, the execution time with conventional registers and call/return mechanism (using the stack) should be determined. For example, if the number of extra cycles (where each *load/store* takes two cycles and all other instructions take one cycle) for each *call return* pair was 50 (saving and restoring registers, passing parameters etc.), *puzzle* and *ppuzzle* would still be faster on RISC. The rest of the benchmarks would be slower. Thus there are some performance advantages to a simple instruction set even without the novel register bank scheme.

Clearly the overlapping register bank scheme is very effective. This is demonstrated by the superior performance for *qsort* and *tower* which use a relatively large number of subroutine calls.

The delayed jump scheme, though probably not critical to the success of RISC, is also quite effective. This is demonstrated by the greatly reduced number of *nop*'s in optimized programs which result in large reductions in the total number of instructions executed (which shows that the *nop*'s are replaced with useful instructions). It should be noted here that the optimizer does other optimizations besides replacing *nop*'s. These other optimizations result in fewer *jump*'s and *ldhi*'s. The effectiveness of the delayed jump scheme is demonstrated by the great reductions in the number of *nop*'s **on top of** the reductions in the number of *jump*'s.

All three of the main features of the RISC architecture are quite effective. The delayed jump scheme is not essential to RISC's success but it does result in faster programs. The fact that RISC is faster for several different types of programs (*puzzle* which has many loops, *sed* which has many memory references and *qsort* and *tower* which have many procedure calls) is encouraging and indicates that the RISC architecture would be a good choice for a general purpose personal computer system.

6. Future Research

A simulator for the RISC architecture, capable of simulating the execution of "real" programs and obtaining dynamic statistics, has been developed. This simulator was used to demonstrate the effectiveness of the RISC architecture.

Clearly, more benchmarks should be run in order to pass final judgement of RISC. The next step is to use the simulator to design an optimal cache for RISC. This would require using the trace analyzer (already written) capable of obtaining cache hit ratios. Another question which should be answered is the effect of the number of register banks in the CPU on RISC's performance. This can easily be studied requiring only slight modifications to the trace analyzer.

The results from the simulation of *tower* show that register banks overflows and underflows may be rather time consuming. This raises the question of what overflow/underflow policy (i.e. how many register banks to store when an overflow occurs and how many to load when an underflow occurs) would minimize the total time they require (this is dependent on the number of overflows/underflows as well as the memory traffic when an overflow/underflow occurs). The policy used to obtain the results in this paper was rather arbitrarily chosen. Studies of this issue can easily be performed using the simulator developed.

Operating system issues have not yet been addressed in the RISC project. Once the operating system features in the architecture are finalized they should be included in the simulator so that some measures on the expected performance of UNIX on RISC could be obtained.

Acknowledgements

This research was sponsored by the Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, and monitored by Naval Electronic System Command under Contract No. N00039-K-0251.

References

1. Campbell, R., *A C Compiler for RISC*, University of California, Berkeley, CA (1981).

2. Campbell, R., *A Peephole Optimizer for RISC*, University of California, Berkeley, CA (1981).

3. Cohen, E. and Soiffer, N., "Static and Dynamic Statistics of C," *CS292R Final Project Reports (unpublished)*, pp. 101-140 University of California, (June 1980).

4. Patterson, D. A. and Ditzel, D. R., "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* 8(6) pp. 25-33 (October 1980).

5. Patterson, D. A. and Séquin, C. H., "RISC I: A Reduced Instruction Set VLSI Computer," *(to be presented) Eighth Annual Symposium on Computer Architecture*, (May 1981).

6. Tamir, Y., *An Assembler for RISC*, University of California, Berkeley, CA (1981).

Appendix 1: Using the Simulator

Generating RISC Object Files from C Programs

Using the RISC compiler and optimizer, the assembly code for any given C program can easily be obtained. If the C program is in file *foo.c*, the assembly code can be written to file *foo.o* by the command:

```
/lib/cpp foo.c | fcom -l | optim >! foo.o
```

where /lib/cpp is the C preprocessor, fcom is the RISC C compiler and optim is the RISC peephole optimizer.

Since a linker loader has not been written yet, all needed routines (except system routines for I/O and dynamic memory allocation) must be compiled together. As indicated in section 3, calls to the system routines are translated by the assembler into calls to "negative" addresses and are later "trapped" by the simulator.

The simulator assumes that the main entry point is address 0. Therefore the C *main* routine must be the first routine in the C source file.

An object file named *foo.out* can be generated from the assembly source with the command:

```
ras -o foo.out foo.o
```

where ras is the RISC assembler.

Simulating the Execution of Programs

The execution of the program can be simulated, using the simulator of the RISC architecture. The command for using the simulator is:

sim [**−a** "command line arguments"] [**−d** maximum data segment size] [**−k** maximum data stack size] [**−w** maximum window stack size] [**−si** standard input file] [**−so** standard output file] [**−se** standard error file] [**−O** register file overflow policy] [**−U** register file underflow policy] [**−S** statistics file] [**−t[d]** trace file] [object file]

The specified "object file" is assumed to contain the output from the RISC assembler. If no object file is specified, the simulator attempts to read file *ras.out*, in the current directory, as the object file.

The options are:

−a "command line arguments"

Allows command line arguments to be passed to the simulated program. The list of arguments should be enclosed in double quotes if *csh* expansion is desired and in single quotes if *csh* expansion is not desired.

−d *n*

The maximum number of bytes in the data segment is set to *n* Kbytes, where *n* is any positive integer. The default is 120.

−k *n*

The maximum number of bytes in the data stack is set to *n* Kbytes, where *n* is any positive integer. The default is 3.

−w *n*

The maximum number of bytes in the window stack (which holds overflowed register banks) is set to *n* Kbytes, where *n* is any positive integer. The default is 7.

—si *file*

The "standard input" of the simulated program is the specified *file*. If *file* is "—" or if this flag is not specified, the "standard input" of the simulated program is the "standard input" of the simulator (which is usually the terminal).

—so *file*

The "standard output" of the simulated program is the specified *file*. If *file* is "—" or if this flag is not specified, the "standard output" of the simulated program is the "standard output" of the simulator (which is usually the terminal).

—se *file*

The "standard error" of the simulated program is the specified *file*. If *file* is "—" or if this flag is not specified, the "standard error" of the simulated program is the "standard error" of the simulator (which is usually the terminal).

—O *n*

The register file overflow "policy" is specified. When an overflow occurs, only *n* occupied register banks will be left in the CPU. 8—*n* register banks will be stored in memory. *n* is an integer between 1 and 7. The default is 4.

—U *n*

The register file underflow "policy" is specified. When an underflow occurs, *n* register banks will be loaded from memory. (Obviously, if the number of register banks in memory is less than *n*, only that number of banks will be loaded). *n* is an integer between 1 and 7. The default is 4.

—S *file*

The simulator collects dynamic opcode frequencies and the number of overflows and underflows and stores the results in the specified *file* at the end of the simulation. If *file* is "–", the results are written to "standard output".

file

The simulator generates an address trace and writes it into the specified *file*. If the **d** option is specified, instruction and data address traces are generated. Otherwise, only an instruction address trace is generated. If *file* is "–", the trace is written to "standard output" for piping into a *trace analyzer*. In this case, if the "standard output" of the simulated program was not directed to some file, it is directed to /dev/tty (i.e. the user's terminal) so that it won't get mixed with the address trace.

Using the Instruction Trace Analyzer

An instruction trace analyzer, called **trace**, has been written. **trace** reads the instruction address, trace produced by the simulator, and the object file, produced by the assembler, and generate various dynamic statistics on the execution of the program. The address trace is either in a file or is piped from the simulator. The latter is usually recommanded since trace files large enough to yield useful results are usually require millions of bytes of disk space.

The command for using *trace* is:

trace [**–i** trace file] [**–j**] [**–p**] [**–r**] [**–s**] [**–o** statistics file] [object file]

The specified "object file" is assumed to contain the output from the RISC assembler. If no object file is specified, *trace* attempts to read file *ras.out*, in the current directory, as the object file.

The options are:

–i *file*

The specified *file* is the instruction address trace produced by the simulator. If this argument is not specified or if *file* is "–", the address trace is read from "standard input".

–j Directs *trace* to produce detailed statistics on conditional jumps. The frequencies of each type of conditional jump and the number of times that each one of these jumps was taken are displayed.

–p Directs *trace* to display opcode pairs frequencies.

–r Directs *trace* to display the number of times each register (actually, each register number) was referenced.

–s Indicates that **static** statistics are desired. No trace file is read and every instruction is assumed to be "executed" once in the order in which instructions appear in the object file.

—o ***file***

The statistics collected are written to the specified ***file***. If this flag is not specified or ***file*** is "—", the statistics are written to "standard output".

Appendix 2: The Implementation of the Simulator

The simulator is written in the C programming language. It consists of about 2200 lines of code residing in eight ".c" (C procedures) files and three ".h" (header) files. All constants and global data structures are defined in header files. Some of these data structures correspond directly to RISC hardware registers. One of the header files ("risc.h") is also used by the RISC assembler. This file includes the definitions of the machine language and a list of "system" subroutines which can be "trapped" by the simulator.

The simulator is implemented by thirty procedures, most of which are used to set up the simulation environment by loading the object file, initializing data structures and handling the "system calls" which are "trapped" by the simulator. The most important routines are:

allocdat	Does dynamic memory allocation for the simulated program. It uses the standard UNIX routines to allocate the memory and then sets up the translation tables used by the simulator to include that memory in the address space of the simulated program.
comargs	If there are any command line arguments for the program being simulated, this routine decodes them and places them in the memory space of the simulated program.
freewindow	Frees the bottom window in the register file by moving its contents to the register window stack.
getwindow	Loads the top window in the register file by loading it with the contents of the top window in the register window stack.
iosetup	Sets up the translation tables between the simulated I/O system and the "real" I/O system.

loadsegs	Loads the object file placing the text segment and data segment in different areas in memory. In order to speed up the simulation, the instructions are stored in memory already partially decoded. In addition, the data stack and register window stack are allocated and the stack pointer is initialized.
main	Decodes command line arguments and calls a sequence of routines which load the object file, initialize data structures, simulate the execution of the program and clean up before termination.
progexit	Terminates execution.
r_cleanup	Cleans up before termination by flushing I/O buffers and closing files used by the simulated program.
regsetup	Sets up the register file and the PC stack.
simulate	Simulates the execution of the program. This routine simulates the operations of the instructions of the program being simulated. It "traps" calls to system routines and calls other simulator routines which perform the desired operations. **Simulate** can also collect statistics and generate address traces.
syscall	Performs the operations of "trapped" system calls. This routine is called with an argument which indicates which system routine was "trapped". The operations of the "trapped" routine are performed with the host computer (the VAX) operating on data structures of the simulated program (as opposed to the usual mode of operation where the data structures of the simulated program are manipulated only by simulated RISC instructions).
tranadd	Translates a given RISC address into the "real" address (i.e. the VAX address) where the desired data is located.

The execution of the simulator begins in **main** which, after decoding command line arguments, calls **loadsegs**, **regsetup**, **comargs**, **iosetup**, **simulate**, **r_cleanup** and **progexit** in sequence.

The routine **simulate** performs most of the functions of the RISC control unit. Instruction fetching and decoding are done by **simulate**. The execution of instructions which do not reference memory, do not cause register file overflow or underflow and do not call "system" routines, is also handled by **simulate** without any procedure calls. An instruction which references memory requires a call to **tranadd**. An instruction which causes register file overflow requires a call to **freewindow**. An instruction which causes register file underflow requires a call to **getwindow**. An instruction which calls a "system" routine requires a call to **syscall**.