AD/A-004 824

EVALUATION OF INSTRUCTION SET PROCESSOR ARCHITECTURE BY PROGRAM TRACING

Amund Lunde

Carnegie-Mellon University

Prepared for:

Air Force Office of Scientific Research
Defense Advanced Research Projects Agency

July 1974

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 75 - 0184	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD/A-004824
4. TITLE (and Subtitle) EVALUATION OF INSTRUCTION SET PROCESSOR ARCHITECTURE BY PROGRAM TRACING		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Amund Lunde		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO-2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE July 1974
		13. NUMBER OF PAGES ~~205~~ 206
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research (NM) 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The thesis develops and evaluates methods for evaluation of the architecture of instruction set processors (ISPs). (An ISP is the logical processor defined by the instruction set, independent of physical implementation). The methods are based on analyzing traces of program executions which contain information about every instruction executed. The main advantages of the methods are:
a) They permit a very detailed study of ISP behaviour. b) They are not restricted to specific languages or processors. c) They are easily programmed.
Methods and experimental results are presented for four aspects of ISP architecture:
(continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. abstract (continued)
register structure, data types and operators, control operators and address calculation. These may be evaluated in terms of four types of costs: execution time, memory space, cost of programming, and the cost of hardware. The methods precented are mostly concerned with time.

A set of programs, the subject set, was used to represent the ISP workload. This was chosen primarily to investigate the variations in the results caused by variation of language, language implementation, algorithm, and programmer.

Register structure is investigated through the concept of a register life. This is the period from when a register is loaded. until its last use before the next time it is loaded. The methods provide data relevant to two problems:
a) What is the optimal number of registers? b) How desirable is generality of registers?

An algorithm is presented which will find how many registers are live at each time during the program execution. This algorithm is extended to compute an upper bound on the increase in time if the program were to run on an ISP with fewer registers. This computation is based on temporarily storing registers that are live but unused for long periods, and on interleaving several lives in one register.

The thesis also presents a classification of the operations that may be performed on a register. This induces a classification of register lives which may be used to assess the need for generality.

Most of the other methods presented apply equally to data operators, control operators, and addressing. The main problems are:
a) How to detect operators that are in the ISP, but not used sufficiently to justify them. This is done by frequency counts and various derivatives therof. Particularly interesting are the frequency results obtained by weighted summation over the whole subject set. b) How to detect oprators that should be included in the ISP. Theis problem is approached by studying instruction sequences.

The main problem in detecting sequences is to reduce the space and time requirements of the analysis program. This problem was solved by using a multi pass algorithm. Each pass extends the existing sequences by one instruction. After each pass, heuristic methods are used to discard insignificant sequences.

The thesis proposes methods to study operand values, information used for control and addressing, information related to the addressing problem for tests, and information on use of indirection.

The most inportant conclusions drawn about the validity of the methods are: The experimental results show good internal consistency. Their trend is independent of algorithm and programming language. They agree well with previous knowledge. The dependence on language is most important for those languages that use a run time system. The use of data operators and data structures depend on algorithm, the register usage does not.

In a subject set for a full scale analysis, the data operators and data structures of the area of applications should be well represented. The individual subject programs should be large enough that dominating loops are avcided.

Evaluation of Instruction Set Processor Architecture by Program Tracing

Åmund Lunde

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
July, 1974

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This work was supported in part by the Advanced Research Projects Agency of the office of the Secretary of Defence (F44620-73-C-0074) monitored by the Air Force Office of Scientific Research; in part by The Norwegian Research Council for Science and the Humanities (Norges almenvitenskapelige forskningsråd). This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

The thesis develops and evaluates methods for evaluation of the architecture of instruction set processors (ISPs). (An ISP is the logical processor defined by the instruction set, independent of physical implementation). The methods are based on analyzing traces of program executions which contain information about every instruction executed.

The main advantages of the methods are:

a) They permit a very detailed study of ISP behaviour.
b) They are not restricted to specific languages or processors.
c) They are easily programmed.

Methods and experimental results are presented for four aspects of ISP architecture: register structure, data types and operators, control operators and address calculation. These may be evaluated in terms of four types of costs: execution time, memory space, cost of programming, and the cost of hardware. The methods presented are mostly concerned with time.

A set of programs, the subject set, was used to represent the ISP workload. This was chosen primarily to investigate the variations in the results caused by variation of language, language implementation, algorithm, and programmer.

Register structure is investigated through the concept of a regisier life. This is the period from when a register is loaded, until its last use before the next time it is loaded. The methods provide data relevant to two problems:

a) What is the optimal number of registers?
b) How desirable is generality of registers?

An algorithm is presented which will find how many registers are live at each time during the program execution. This algorithm is extended to compute an upper bound on the increase in time if the program were to run on an ISP with fewer registers. This computation is based on temporarily storing registers that are live but unused for long periods, and on interleaving several lives in one register.

The thesis also presents a classification of the operations that may be performed on a register. This induces a classification of register lives which may be used to assess the need for generality.

Most of the other methods presented apply equally to data operators, control operators, and addressing. The main problems are:

a) How to detect operators that are in the ISP, but not used sufficiently to justify them. This is done by frequency counts and various derivatives thereof. Particularly interesting are the frequency results obtained by weighted summation over the whole subject set.

b) How to detect operators that should be included in the ISP. This problem is approached by studying instruction sequences.

The main problem in detecting sequences is to reduce the space and time requirements of the analysis program. This problem was solved by using a multi pass algorithm. Each pass extends the existing sequences by one instruction. After each pass, heuristic methods are used to discard insignificant sequences.

The thesis proposes methods to study operand values, information used for control and addressing, information related to the addressing problem for tests, and information on use of indirection.

The most important conlusions drawn about the validity of the methods are: The experimental results show good internal consistency. Their trend is independent of algorithm and programming language. They agree well with previous knowledge. The dependence on language is most important for those languages that use a run time system. The use of data operators and data structures depend on algorithm, the register usage does not.

In a subject set for a full scale analysis, the data operators and data structures of the area of applications should be well represented. The individual subject programs should be large enough that dominating loops are avoided.

ACKNOWLEDGEMENTS

First and foremost my thanks go to my advisor, Professor William A. Wulf, for encouraging the project in its early stages, for contributing time and ideas throughout, and for reading innumerable versions of the draft.

I am further indebted to my thesis committee, Professors John Grason, A. N. Habermann and Daniel P. Siewiorek, and also to Professor Mary M. Shaw and Dr. Mario R. Barbacci. They all suggested significant improvements to the presentation.

Credit also goes to Ric Werme who helped debug my additions to his tracer, to Bill Wulf, George Rolf, Larry Flon and Mario Barbacci who each programmed a version of the Aitken algorithm, and to Rich Johnsson who helped out with the document production system.

The Computer Science Department at Carnegie-Mellon provided a stimulating environment for both research and relaxation.

Final thanks go to CMU-10A: object and tool, lab assistant and analyst, editor and typesetter, plague and playmate. Without her support, this work would never have been completed.

A NOTE ON TERMINOLOGY

By an _instruction set processor_ or _ISP_ we mean the logical processor defined by the instruction set, as opposed to its physical implementation. Included in the ISP structure are such things as instruction formats, register structure, instruction interpretation algorithm (including address calculation), datatypes and their representation, etc. Computer families, like the IBM 360 and 370 series and the CDC 6000 series are examples of ISPs with several different physical implementations.

Obviously the logical structure can not always be entirely divorced from its physical counterpart, nor is such a separation always desirable. There should be no doubt, in our further discussion, when we take the physical aspects into account.

We use the term ISP to mean the instruction set processor itself, not the notation for describing such processors defined by Bell and Newell ([BelC71]). As a concession to readers unfamiliar with it, we have tried to avoid using this notation. The associated terminology, however, is used.

Italics are used for words that are previously defined. Underlining is used for words that are being defined, or otherwise stressed.

In the tables of results, 0 means an exact zero, 0.000 or similar constructs mean less than 1/2000 (in this case) but not exactly 0.

Unless otherwise stated, the term "PDP-10" is used to mean the DECsystem10 ISP or the KA10 processor of that system, both described in [DEC71].

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

> Speilet er givet os Mensker av Gud
> men Fanden har giet det den Feil:
> at det aldrig kan vise hvordan man ser ud
> naar man ikke ser i et Speil
>
> Kumbel Kumbell

This thesis is concerned with the architecture of Instruction Set Processors. It identifies the most important parameters of such architectures, their interdependence and their associated costs. It proceeds to present a collection of methods for evaluating some of these costs. Most of the effort of the thesis lies in developing these methods and studying their performance for one ISP and a set of programs (a subject set) running on that ISP.

Our point of view is that of the programmer, or maybe more correctly, that of the program being executed. The goal of our methods is to evaluate the features of ISPs in terms of their utility to the program (or programmer). Thus the questions that they will attempt to answer can be generalized to: "How well does the programmer/compiler utilize the features made available to him through the instruction set? Which of these features should be removed or changed? Which should be added?"

The methods are based on analyzing traces of programs being executed, where the trace contains information about every instruction executed by the program. The analysis is performed by separate programs, and is thus completely disjoint from the writing of the trace. Most of the methods presented, and certainly the most important ones, have been implemented as programs and used in experiments. The experimental results agree well with previous knowledge and with intuition, and are also consistent among themselves. Hence the experimental evidence supports the validity of the methods.

The experimental results that we present are from experiments designed primarily to evaluate the methods, not the ISP that we have worked on. In particular the programs we have analyzed are small, and from a restricted application area. Hence, although many of our results certainly permit valid conclusions about the ISP we have worked with†, our set of subject programs has been too restricted to provide the basis for a valid, full scale evaluation of a general purpose ISP.

- - - - - - -

† The PDP-10

1.1 Overview of the thesis

This introductory chapter presents an overview of the basic ideas of the methods. It then gives a survey of related work and relates our work to this.

In Chapter 2 we present the types of cost associated with implementing and using (or not using) ISP features, and discuss their relationship.

Chapter 3 describes the major sources of errors and variation that might influence our experimental results, and describes how we selected a set of subject programs to evaluate these influences.

Chapters 4 through 7 contain the core of the thesis. In those chapters we analyze the instruction set processor, concentrating on those features for which we have developed methods of evaluation. The order of presentation is:

Chapter 4: Register structure
Chapter 5: Data types and their operators
Chapter 6: Control operators
Chapter 7: Address calculation

Each chapter is further divided into sections, each discussing a different feature or aspect of the chapter topic. For each feature, we discuss the motivation for having this feature, and the costs and tradeoffs associated with it. Our methods for estimating some of these costs are described, and experimental results are presented where applicable. For each method its limitations, sources of errors, and dependencies on the various sources of variation, as presented in Chapter 3, are discussed.

For our analysis we rely heavily on the multidimensional computer space presented by Bell and Newell [BelC71]. The dimensions of this space represent such things as intended application, technology, word size, etc., and possess several levels of detail. We have made this structure finer or coarser to suit our needs, and will use it freely below without further reference to its origin.

The most important dimensions for classification of instruction set processors are (with those most highly related on the same line):

Computer (system) function
Processor function
Memory accessing algorithm - primary memory size
Addresses per instruction - M.processor state
Word size - number base - data types
Control structures

As stated in Section 3.1, we take the computer and processor functions to be given, i.e. we investigate general purpose computers with a bias towards scientific calculations. The next four coordinates above each corresponds to one of the four chapters listed.

The last chapter summarizes the results and points out areas for future research.

The thesis describes two processes more or less in parallel. One is the development of the methods and their use to evaluate ISP architecture in terms of the costs discussed in Chapter 2. The other is the evaluation of the methods themselves, in terms of the framework described in Chapter 3. Both processes go on through Chapters 4 to 7, and conclude in Chapter 8.

1.2 The problem

Several approaches may be used to improve the performance of computers. These are to a large extent orthogonal and are often combined, as exemplified by many current commercial designs.

One approach is to use faster circuit technology for a brute force increase of speed, leaving the ISP architecture unchanged. This approach is of no interest to the present discussion.

Another approach involves radical changes in the organization of the central processor, in particular higher degree of parallelism on the task, instruction or sub-instruction levels. This sometimes implies more or less drastic changes in the way programs are thought about and formulated, as exemplified by the CDC STAR [HolS71], ILLIAC-IV [BarG68], and C.mmp [WulW72] machines. In other cases, as in the CDC 6600 design [ThoJ64], parallelism is on the instruction level, retaining the classical instruction stream concept and at worst requiring local reformulation of the algorithms. Instruction parallelism is peripherally of interest to our discussion, (see Section 2.3). Parallelism on the task level is outside the scope of this thesis.

A third approach is to improve the architecture of the Instruction Set Processor (ISP), but staying within the classical Von Neumann type of machine. This approach is the background for our work. A difficulty with it, but also a major reason for it, may be the interest vested in existing instruction sets. In such a case the problem may be how to extend it compatibly, or to find features that may be removed at a reasonable cost. Data provided by our methods may be used in solving this problem and also to some extent when designing new instruction sets from scratch.

There is ample evidence that the ISP architecture is indeed an important factor in processor efficiency and economy. Notable is a study by J. A. Stewart [SteJ.nd], comparing program sizes and execution speed of three contemporary computers[†] having approximately the same word sizes[††] and instruction execution times[†††]. When moving benchmark programs between these computers, program sizes varied by factors from 1.3 to 2.7 and running time by factors up to 5[††††]. Some of this variation may be due to inferior compilers and other software. However, code sequences for commonly occuring constructs indicate that the problem to a large extent lies with the instruction set.

Another example is provided by the Burroughs B1700 computer, (see page 15). A considerable gain in space and time is claimed by the designers of this computer system, achieved by designing instruction sets tailored to the higher level language used.

Human intuition about program behavior is notoriously bad. This has been demonstrated by several investigators. One example is given by Knuth in his well known study of FORTRAN programs [KnuD70]. The personal experience of people who have observed some aspect of their programs' behavior, as reported in countless stories of computer folklore, tend to corroborate this.

The cited studies clearly demonstrate a need for quantitative methods which can aid the ISP architect in deciding values for the design parameters of his ISP, and to justify his decisions. The data obtained should be as independent of technology as possible, so that they will not change as technology progresses. They can then be used to compare the cost of implementing a structure using different technological solutions, or to compare the cost and utility of different structures in the context of the available technologies.

- - - - - - -

† The IBM 360/44, the SDS Sigma 5 and the PDP-10.

†† 32 or 36 bits.

††† For commonly used instructions, factors ranged from 0.7 to 1.8 compared to the PDP-10.

†††† The PDP-10 being the best

Ideally the behaviour of all programs executing on the ISP should be studied. This can be done only superficially, as by accounting data and similar information. For a detailed study one is forced to restrict oneself to a set of, hopefully representative, subject programs. Given an application area, and such a subject set to represent it, there are several methods of obtaining data on program behaviour. They may be classified as static or dynamic methods, depending on whether data are collected before or during execution.

Static information can be collected manually, by compilers, or by some program analyzing the relocatable or absolute code. Such methods should be used to obtain the space cost (see Section 2.3) of the code and static data structures, but can not be used to obtain information pertinent to the execution behavior of the subject program. For this purpose dynamic data are needed. Several methods of obtaining such data are described and compared in Section 1.2.1. We chose to use traces containing information on every instruction executed by the program. These traces are written on an appropriate storage medium, and are analyzed later by separate programs. The advantages of this method are that the exact sequence of events is preserved, and that a large amount of detail may be recorded. We discuss the appropriateness of this choice in Section 2.

As we present the methods, their intended domain is to evaluate the features of ISP architecture. The particular ISP design parameters that we consider include the number and types of registers, the data types and their operators, control operators and their associated data structures, and address calculation methods. Our methods fall mainly in two groups, one dealing with register structure, the other with data and control operators.

Register structure is evaluated through the concept of "register lives". We present a method to detect such lives, and to find to what extent registers are simultaneously alive. From this we are able to find an upper bound on the increase in execution time which would follow if the number of physical registers were reduced. We also present a method to assess the need for generality of registers.

Our methods for operators and data types are based on frequency counts of single operators and of sequences of operators. We present an algorithm for counting the occurrences of sequences of arbitrary length, including a set of pruning heuristics designed to detect which sequences are in some sense significant. Only occurrences of such sequences are counted; this is what makes our algorithm economically feasible.

We expect the methods to provide useful evaluation of existing designs as well as suggest improvements in existing designs and give ideas and guidelines for new designs. Such new designs could be for general purpose processors, or for processors specially designed for some particular language or some special class of computations. Such a specialized application is defined more by the selection of subject programs to which we apply our methods than by the methodology as such.

Our methods can also be applied to domains less related to ISP design. As will be seen they have obvious applications in compiler design and language design, and also in the art of tuning programs to make them more efficient. In particular we expect our method for register utilization to be of interest to these domains.

As in any other inquiry, the answers to one set of questions raise new questions that one would like to answer. In some cases our methods will produce compact data bases which will allow certain kinds of simple questions to be answered after the original analysis, and at a much lower cost.

1.2.1 Obtaining dynamic information

Dynamic information can be collected by hardware monitors, by programs running in parallel with the subject program†, by code inserted into the subject program by the compiler, or as in our case, by running the subject program on an interpreter for the ISP in question. In any case, the data can be analyzed on the fly or saved for later analysis by special programs.

Programs or hardware monitors may be used to sample the program counter and other pertinent parts of the processor state. This can give us information about the (relative) frequencies of various events, such as the execution frequency of the different parts of the program. Considerable analysis of the subject program is required to obtain information about its local behavior. Information about the sequence of events, such as the behavior across programmed jumps, can not be reconstructed completely. Also no information about register content and operand values is available. Furthermore, in the case of sampling by program, the results are not exact, but depend on sampling rate and random events.

Code inserted by the compiler is usually restricted to maintaining execution frequency counts

----

† As can be done in several contemporary systems.

for each straightline segment of code, since collecting more extensive information this way would make code size prohibitive. Hence we again have the problem of reconstructing sequences of events. Considerable analysis is needed to obtain detailed information on the ISP level behavior of the program, since the primary data relates to the language level. We are furthermore restricted to analyzing programs written in languages that have this feature in their compiler (or a suitable preprocessor), and which are available for recompilation. It also disturbs locality aspects of the program execution. It is, however, more accurate than sampling, since we are guaranteed that all executed parts of the code are represented in the results in proportion to their execution frequency.

We chose to run the subject program using an interpreter for the ISP under investigation, and collected information on each instruction as it was interpreted. This method is usually called instruction tracing, or just tracing. The information was, in our case, written on magnetic tape. This method allows one to study not only the instruction stream as seen by the processor, including the path taken through sequences of programmed jumps, but also to follow operand and index values, indirect address chains etc., if so desired.

Also, tracing is language and compiler independent. It can be applied to any subject program that can be brought into the format acceptable to the interpreter. In many cases (as in ours) the interpreter will be a relocatable module running on its own ISP, which will then accept the standard relocatable format for the subject program. For a microprogrammed processor, the microprogram may be extended to output the information desired (See page 16).

A further advantage is that analysis is naturally separate from the data collection. Provided a rich enough trace is written, new types of analyses can be performed at any time without having to retrace the subject program. Since writing the trace is cheap compared to analysing it, this may at first sight seem to be of little value. It does, however, guarantee that the results of different analyses are consistent and independent of changes in the program traced, the compiler compiling it, and of random environmental influences.

In terms of computer resources needed to apply the methods, tracing is probably more costly than the others. Tracing a program using our current interpreter† increases running time by a factor of about 60, and the analysis programs are slow. This is, however, of little importance. As will be seen, a considerable amount of detailed information can be obtained at a cost which is not prohibitive, and the writing of the analysis programs is straightforward compared to what it would be with the other methods, to obtain similarly detailed information.

\- - - - - - -

† Interpreting the PDP-10 on the PDP-10

To have sufficiently detailed information, we wrote at least 4 words of trace for each instruction executed. These were: The instruction word, the program counter and effective address, the contents of the accumulator and of the effective address. If indirection or byte access was used, two further words were written for each level of indirection, containing the address and contents of the bytepointer or indirect word. Writing at 556 bpi and blocking 1000 words to a tape record, this allowed us to trace about 600 000 instructions on a 2400 ft. reel of tape. This corresponds to 1.5 - 2 seconds of CPU (PDP-10/KA10) time when executed at full speed.

Most of our methods use only the instruction word. Hence time could be saved both while tracing and analyzing, by omitting the other information in the trace. This would also permit more information to be written on each tape. In the interest of generality, however, we used the approach stated.

An alternative to instruction by instruction tracing is the jump trace described by Alexander [AleW72], (see page 14). With this tracing method information is written to the trace only at instructions which change the program counter. In between such points the program runs at full speed. This method is fast, but information on operands and register contents between tabulation points is lost. To fully realize the gain in speed, the compiler should know about the tracer and insert appropriate instructions to call it. Analysis is simplified if the compiler also outputs a file of descriptions of each straightline segment of code. This dependence on the compiler restricts the set of subject programs that can be analyzed, increases code size and disturbs locality, as discussed above.

1.3 Restrictions in domain

We will restrict ourselves to traces obtained by executing single programs on an interpreter for the ISP to be evaluated. This means that we bar ourselves from studying problems related to interrupt handling, detailed I/O management, multiprogramming and other operating system issues. On the other hand it allows us to concentrate on the behavior of one single program during a continuous span of time, without being disturbed by interference from other programs. This permits a study of the local behavior of the subject program to any desired level of detail. From this point of view the invisibility of interrupts is a strength rather than a restriction. Also, a change in the execution speed of an operating system will imply a change in the behaviour of its environment. Hence in studies of operating system behaviour one should restrict oneself to information that can be collected on the fly.

A further advantage is that the trace is reproducible and free from random perturbations caused by interrupts etc. This is not strictly true for programs that use shared resources (such as primary memory dynamically allocated to users) or resources that operate in parallel to the traced program. In such cases different code might be executed depending on resource status.

Although most of our methods are applicable with minor modifications to most ISPs, we focus our attention on ISPs with a general register structure. We take this term in a wide sense, meaning roughly that a sizeable repertoire of operations is available uniformly over a vector of 4, 8 or more registers. Another characteristic is that the registers can be addressed from more than one field of the instruction word†. (See also Chapter 4). Limiting cases are 2 or 3 address machines on one hand and one address machines with no index registers on the other; we do not, however, consider these.

Our experimental results are from the PDP-10, which has a vector of 16 extremely general registers, and a very general instruction set, particularly for control operations (a rich set of skips and jumps, several forms of subroutine jumps etc.). Hence this ISP is a good starting point for detection of unnecessary features. However, as will be seen, we have also been able to detect some deficiencies of this ISP that are not due to unnecessary generality.

1.4 Related work

Studies of frequency counts of instruction executions have been described by several authors. The best known is the Gibson mix, developed by Jack C. Gibson at IBM in 1959. Gibson divided the instructions of the IBM 704 and 650 into 13 classes and counted how many instructions were executed from each class. His sample size was 17 programs, approximately 9 million instructions. The results are described in [GibJ70]; we tabulate them in Figure 5-3.

Gonter [GonR69] has compared the Gibson mix and the UMASS mix ††, using essentially the same classification and tracing 15 million instructions on the CDC 3600. His results correlate well with Gibson's; they are tabulated in Figure 5-3.

- - - - - - -

† Accumulator field, index field, memory address field, base register field etc.

†† UMASS = University of Massachusetts

The substance of these results is that LOADs and STOREs account for about 30% of the instructions executed, branches for 16% to 38%, index manipulations 13% to 18%, arithmetic 3% to 19%. The results depend both on the ISP and the subject set.

Other similar mixes and experiments are reported by Arbuckle [ArbR66], Conners, Mercer and Sorlini [ConW70], Raichelson and Collins [RaiE66], and Herbst, Metropolis and Wells [HerE55]. The latter is the earliest report known to the author.

The emphasis of the above studies was mostly on evaluation of the raw processing capacity of the central processor. Little emphasis was made on improvements in the instruction repertoire or central processor structure.

Foster, Gonter and Riseman, [FosC71a] have gone one step further, by starting to investigate the effects of reducing the instruction set. They report their experience with two measures of instruction set utilization. Both of these measures are equally applicable to static and dynamic instruction counts. The static measures give an estimate of the space cost (Section 2.3) and the dynamic measures estimate the time cost (Section 2.2) associated with using the instruction set. The examples of [FosC71a] use the CDC 3600. Our use of these measures is described in Section 5.1.

The first of their measures is the undiluted information-theoretic measure of information content:

$$I = - \sum_{i=1}^{T} p_i * \log2(p_i)$$

where

p_i is the probability of using the i'th opcode

T is the total number of different opcodes

log2 is the logarithm base 2

Intuitively, the interpretation of I is the average number of bits of information conveyed by each opcode. The value of this measure is doubtful, particularly with a fixed wordlength, since the space that could be saved in each instruction word by using the encoding depends on the frequency of occurrence of the instruction in question, and has no relation to its need for operand addressing capability etc. Furthermore, optimal encoding with respect to it implies variable length encoding of the opcodes and a correspondingly more complicated decoder[†].

The other measure they propose is a function computed as follows: Order the operation codes by frequency of occurrence. Let C_i be the number of occurrences of the i'th opcode in this ordering, ($C_i \geq C_{i+1}$ for all i). Let P be the total number of instructions in the sample, and T the number of different opcodes, as before. The FGR function is then computed as:

$$FGR(N) = 1 - 1/P \sum_{i=1}^{N} C_i \qquad (1 \leq N \leq T)$$

This function measures the effort necessary to recode or run the original program on a central processor with a smaller instruction set. Indeed FGR(N) is that fraction of the instructions which would have to be recoded (static) or interpreted (dynamic), were the instruction set reduced to the N most commonly occurring instructions. For some of these the recoding might be impossible, this is not taken into account.

Substituting execution times for C_i and P above, and ordering the C_i accordingly, we obtain a measure of the fraction of execution time accounted for by the omitted instructions, in this case the least timeconsuming ones.

These measures were used on a set of CDC 3600 programs. In the dynamic case the suboperation field of the opcodes was disregarded. Also, a different sample was used for the static results than for the dynamic ones. The static I varied from 3.59 to 5.36 for the different programs, with a theoretical maximum of 7.16. The dynamic I varied from 3.94 to 4.64, with a theoretical maximum of 6.00. FGR(32) varied from 0 to about 0.2 in the static case, and from 1 to 0.06 in the dynamic case. This shows that a reduction of the instruction set to 32 instructions would cause some increase in program space, but that the instructions that must be interpreted are ones that are executed rarely.

A related study is by Foster and Gonter [FosC71b]. They investigated the effect of interpreting opcodes differently depending on the recent history of the ISP. Thus on a one accumulator machine the sequence LOAD ADD occurs often, LOAD LOAD hardly ever. Hence the LOAD and ADD instructions might use the same encoding in the instruction word, provided the LOAD instruction changes the state of the decoder. A "set state" instruction provides the necessary escape mechanism. The intended application is to combine a large instruction set

----

† An approximation to this encoding was used with the Burroughs B1700. See further discussion on page 15.

with a small opcode field, thus freeing instruction word space for addressing. They verify their idea by an analysis of some CDC 3600 programs.

The results show that over 67% of the instructions could be executed without use of the escape mechanism, even if the opcode field was reduced to 3 bits. For a 5 bit field, 95% of the instructions could be executed directly. By circumventing some machine specific properties in their data, the result for 3 bits was improved to 74%.

Riseman and Foster [RisE72] [FosC72] have used traces to study the effect of data dependencies on the execution speed of parallell processors. They postulate a machine where only the execution of the instructions take time; instruction fetch and dispatch, and data fetch and store, take no time. Further there is an infinite supply of registers and functional units so that no instruction is held up for the lack of hardware. The instruction set is as for a CDC 3600, and traces from this machine were used in their experiments.

There are two restrictions which prevent instructions from being executed:

a) Their operands have not yet been computed.
b) The exact instruction to execute can not be determined until some condition (jump) has been resolved.

Restriction b) can be circumvented by assuming a nondeterministic processor, where both paths of the program are executed in parallell until the condition is resolved. This nondeterminacy can be carried to infinite depth, or restricted to a maximum of N unresolved conditions.

The experiments show an average speedup by a factor of 1.72 for $N = 0$, 2.72 for $N = 1$, 7.21 for $N = 8$, and 24.4 for $N = 128$. For infinite nondeterminacy ($N = \infty$) the speedup was by a factor of 51.2. Similar results were found by Tjaden and Flynn [TjaG71]. The results show that conditional jumps, and their dependency on calculated results, is a severe restriction on execution speed.

Several investigators have used traces to study addressing patterns, with the object of determining optimal design of paging systems and cache memories. We mention Coffman and Varian [CofE68], Gibson [GibD67], Hatfield [HatD72], Kaplan [KapK71] (see below), Lewis and Yue [LewP71], and Seligman [SelL.nd].

A few authors have described more comprehensive studies based on traces:

At IBM, Murphey and Wade [MurJ70] used traces to evaluate the performance of the IBM 360/195. Traces were made of programs believed to be representative of the 195 workload, as they were executed on other 360 models. Detailed studies were made of the behavior of these programs in a 195 simulator. The emphasis of this study was on design validation and performance prediction. Particular studies were made of the efficiency of the mechanism for parallel execution of different instructions.

Winder at RCA [WinR71], [WinR73], describes the method of tracing used on the RCA Spectra 70/45 and also in some detail the various studies performed. These include cache system studies [KapK71], paging analysis, miscellaneous program statistics emphasizing I/O, branching and conditions, indexing, and operand length for variable length operands. A SIMSCRIPT simulator driven by the trace was used to investigate architectural variants like memory banking, cache parameters, instruction lookahead, multiprocessing etc.

Wortman [WorD72] has designed an experimental technique to evaluate computer architecture, in particular its suitability for particular programming languages. It is based on collecting static and dynamic statistics on the use of language fragments. Language fragments are constituents of program code which map into non-overlapping segments of object programs, and which do not contain data dependent loops. As a case study Wortman chose a PL/I dialect called Student PL, and designed a stack oriented architecture suitable for this language. An interpreter for the architecture was written, and also a compiler to translate Student PL programs into its machine language. For his subject set he chose about 1000 small student programs from an undergraduate programming course. Three kinds of statistics were observed:

Source program statistics, essentially the number of application of each production during syntax analysis.

Object program statistics, i.e. frequencies of occurrences of the machine instructions (language fragments), and pairs and triples of these in the generated code.

Run time statistics, i.e. frequencies of execution for the individual machine instructions.

Based on these statistics he made several improvements in the instruction set, and found reductions of about 50% in each of program storage space, data and instruction accesses, and number of bits accessed. The most significant improvements were:

Information relating object instructions to source lines was moved to secondary storage.

The data accessing method was improved.

An immediate type instruction was introduced to move constants to the stack. (72% of the constants found were integer constants, and 98.8% of these could be represented in 6 bits).

The handling of conditionals and "builtin" functions was improved.

By refining his language fragments Wortman also was able to compare his machine design with the IBM 360 as a vehicle for PL/I.

Alexander [AleW72] has made a study similar to Wortmans, but for an excisting ISP (The IBM 360) and a language (XPL [McKW70]) used mostly for compiler writing. His main goal was to investigate how the features of the XPL language were used, and what requirements they posed on the ISP. He presents statistics on source programs, object programs and run time behaviour. These were obtained by modifying the XPL compiler (XCOM), and by full tracing and jump tracing. His subject set was slightly different for the different analyses, it consisted of XCOM, several compilers written for undergraduate and graduate courses, and his own analysis programs. His results can be summarized as:

Floating point and decimal arithmetic are not used by XPL, this leaves 91 instructions that can potentially be generated by XCOM. Of these only 47 were actually generated. 10 of these account for 84% of the instructions executed. The 10 most generated instructions account for 85% of the total number of generated instructions, this set intersects the previous set of 10 by 9 instructions.

XCOM allocates 3 registers as accumulators. The first of these was named in 47% of the accumulator references (as opposed to index or base register references). The second was named in 26%, and the third in 11% of the accumulator references. Hence expressions rarely are complicated enough that many accumulators are needed. The register used for indexed access accounts for 11% of the accumulator references.

42% of the references to index or base registers were to register 0, i.e. no indexing or base was used. That is: almost half of the addresses were unmodified. 8% were used in array accessing, 31% were used to access statically allocated data (as base). 7 fixed registers were allocated by XCOM for this latter purpose.

Most of the branches were to locations close to the branching instruction. Alexander suggests that the branch instruction of the 360 could be modified to address relative to the current program counter, and the 4 bits now used for base register addressing could instead be used to augment the written address field, to make it 16 bits long. Such an instruction would suffice for 99% of all branches. 5K bytes of load instructions would be eliminated, saving 15% of the program space.

If opcodes were conditionally decoded, as proposed by Foster and Gonter [FosC71b] (see above), 16.2% of the program space could be saved by an encoding of the opcode in 3 bits. This result pertains to one particular subject program.

Alexander extensively compares his dynamic and static results, and comments upon the significance to constructs used or not used within loops, and on special properties of the XPL language and system. He also advocates the use of program profiles, and in this context points out the need for string manipulating instructions in compilers.

Studies of architecture based on tracing have probably also been performed by computer manufacturers. Such work is usually considered "company private", and is not published, but a few have been: The work by Murphey and Wade [MurJ70], and that by Connors, Mercer and Sorlini [ConW70], all at IBM, and also that by Winder [WinR71], [WinR73] and Kaplan [KapK71] at RCA. All of these are mentioned above.

A particularly interesting machine design is the Burroughs B1700, [WilW72a], [WilW72b]. In this system microcoded interpreters are provided for several "S-languages", each of them corresponds roughly in level to a classical machine language, but is tailored to fit the needs of a particular higher level language. The microprograms address memory by bit position, and desired access width is supplied on each access. Hence the processor gains efficiency primarily in two ways:

a) Time efficiency is gained by using an S-language tailored to the application (higher level language), hence having essentially the "right instructions" for the task at hand. Each instruction is usually more complex than most classical machine instructions.

b) Space efficiency is gained by encoding the S-language instructions in different formats depending on the need for space to represent the feature in question, and its frequency of use.

One such S-language is SDL, particularly suited to systems programming. The opcodes of this language are of 3 lengths, 4, 6 or 10 bits, whereas a fixed length encoding would require 8 bits. By using this encoding, space is gained at the cost of an increased decoding time. The two encodings mentioned were compared to the Huffman encoding, which is space optimal. The following results were found:

Encoding:	Space saved:	Time lost:
Fixed 8 bits	0%	0%
SDL 4, 6, 10 bits	39%	2.6%
Huffman code	43%	17.2%

Hence the chosen encoding is almost as space efficient as the Huffman encoding, and almost as time efficient as the fixed field encoding.

Similarly the SDL addresses were encoded using 8 different formats and a 3 bit field to distinguish them, giving a 38% saving in memory space compared to the 4 byte addresses needed on a byte oriented machine with fixed length addresses spanning the same address space.

For FORTRAN and COBOL programs, using the appropriate S-language, the reduction in program space was found to be 40% - 70% over the IBM 360 and the Burroughs B3500.

Furthermore, access width can be a parameter to the S-language interpreter, allowing the compiler to generate code more suited to the actual problem and also making possible a planned "Dial a precision FORTRAN".

Wirth ([WirN72]) has given a qualitative review of a particular ISP, the CDC 6000 series, from the viewpoint of programming ease and error detection. In particular he points out deficiencies of the data representations and operator implementations that make the detection of errors, and hence the guarantee of a correct result, impossible or at best uneconomical. He also points out the lack of an instruction for calling reentrant programs. His experience is from the implementation of PASCAL [WirN71] for this ISP, but his arguments apply equally well to all language implementations where security and error detection is a design goal, and to all uses of recursion or reentrancy.

For microprogrammed processors, the microprocrammed interpreter can be extended to collect execution time data. This approach is advocated by Saal and Shustek [SaaH72]. For simple types of data this allows the subject program to run at almost full speed. However, full tracing by microprogram will be limited in speed by the device recording the trace.

Since analysis time is considerably larger than trace time in any case, the advantage is doubtful. The authors discuss various aspects of implementing such techniques, and present data relating to opcode utilization and frequent instruction pairs. These results differ little from those of Alexander [AleW72] and Foster et. al. [FosC71b].

We have previously identified the most important dimensions of ISP architecture to be: register structure, data types and operators, control operators and structures, and address calculation.

Of these, the operator dimensions have been relatively well explored in the works cited. This applies in particular to studies of the utilities of existing operators and possibilities for more efficient encodings. The problem of finding desirable but non existing opcodes has been touched upon by Alexander and Wortman, but needs further work.

Other properties of control have been partially explored, particularly locality of jumps (Alexander), and the use of test instructions and conditions (Alexander, Winder). Locality properties of address streams have been studied in connection with virtual memories and caches, but the data structuring aspect is largely unexplored. Register structure has barely been touched (Alexander).

1.4.1 Contributions of the thesis

Our main contribution to this field of work is the methods for register utility and generality.

We also break new ground in our work on instruction sequences. Previously Alexander (see page 14) has presented dynamic counts of sequences, but only of length up to 3. Our present program can accumulate counts for sequences of lengths up to 20†. Our pruning heuristics make the accumulation of counts for sequences of this lenght economically feasible. In fact we point out an improvement to our algorithm which will make the accumulation of sequences of this length and longer much more efficient than with our present program.

Finally our approach is general (see Section 1.2.1), we present results spanning algorithms

- - - - - - -

† This limit was arbitrarily set because we believed longer sequences would not be of interest. The method can handle sequences of arbitrary length.

coded in several languages and by different programmers, and we try to evaluate the influence of these factors on our results. Earlier work has in some cases ([AleW72)] and [WorD72]) been confined by methodology and other considerations to one language. In other cases the selection of subject programs and goals have been more restricted.

We can not leave this section without mentioning the influence on our work by that of Foster, Gonter and Riseman [FosC71a]. The FGR function introduces some very simple and relevant measures of the utility of ISP features, namely the change in execution time or instruction count resulting from a change in the ISP. Foster et. al. applied this idea to opcode utilisation. Much of our work consists of applying it to other features of ISP architecture.

CHAPTER 2

COSTS

In this chapter we discuss the various basic cost measures pertaining to ISP features. After some introductory remarks we list four types of cost. For each of these we discuss its definition and other relevant issues, such as the way or ways we measure it and their related inaccuracies, other ways to measure it, and its relation to the other types of costs. As a necessary introduction to this discussion we will make some comments on the instruction word and issues related to it. This follows after the introductory remarks.

The four types of cost we propose are general. We believe they apply to all ISP structures, not only those with general registers. The units in which we measure might, however, vary with the structure of the processor in question. This is true even within the class of general register processors.

Computer resources are allocated in units of space and time: space in memory units, time in processing, control and communication units. Since some memory must be in use whenever the central processor is in use, the product of space and time is a relevant measure of cost for the usage of memory units and time alone for other units. These are the basic units for measuring the costs incurred by running the program on the machine. Relating these to economic terms requires knowledge of the actual cost of the units of the computer, and of the operating expenses. In addition, the cost of producing the program (designing, coding and debugging), in terms of human effort and machine resources, depends on a good ISP design and may be highly relevant.

Since we are concerned with the ISP we will disregard costs related to secondary memory except insofar as they are expressed by the costs relating to primary memory. Similarly the basic instructions for I/O are not part of the ISP seen by the user (See Section 1.3), hence we also disregard I/O costs and the costs of control and communication units. The latter are to some extent expressed by the cost of the central processor. The time cost (see below) associated with I/O and secondary memory usage is considered independent of and irrelevant to ISP architecture, and will be disregarded except where explicitly noted otherwise.

Motivated by the above remarks and by further discussion below, we will regard the costs of having or lacking a given feature in an ISP as falling in 4 basic categories:

1) Execution time (time cost)
2) Memory space (space cost)
3) Programming effort (programming cost)
4) Hardware to implement the feature (hardware cost).

This list is roughly in order of importance. Our methods will be almost solely concerned with time cost, but the others will be kept in mind and mentioned when relevant.

The weighing and trading off of these costs is the concern of the ISP designer and falls outside the scope of this thesis. Our goal is to provide methods for computing them, and in particular the time cost, exactly or approximately, as seems relevant and possible for the feature in question.

2.1 The role of the instruction word

The instruction word occupies a central position in any ISP design, being the quantum in terms of which the ISP forces the programmer to express his algorithm. Hence it brings together all the issues of ISP design and must be a focal point for our research.

Some different views on how the instruction word can be organized are represented by the CDC 6000 series, the PDP-10 and the IBM 360 series. The 6000s have 60 bit words and about 70 different user instructions packed 2 to 4 to a word; the PDP-10 has 36 bit words and about 420 different user instructions each filling one word; the 360 has about 130 user instructions of 16, 32 or 48 bits, the major data formats are 16 or 32 bits, memory fetch width is 8, 16, 32 or 64 bits depending on the model. Good performance is attempted in the first case by fast instruction issuance, in the others by powerful instruction sets.

We now present some of the issues relating to the instruction word organization in a top down order, neither implying any order of importance nor a sequence in which design decisions should be made. As is exemplified by the above designs, there is no generally accepted way of resolving these issues. In fact, the solution is often strongly influenced by historical or marketing constraints, or other external considerations. In particular the introduction of the 8 bit byte by IBM with the 360 series in 1964 has had a standardizing influence.

The first issue is the size of the instruction word. The cost and power ranges, and in particular the addressing space, planned for a new processor, will to a large extent influence what features need to be accommodated in the instruction word. Its size is also influenced by issues not relating to the instruction word as such, particularly the desired accuracy of the arithmetic and other data types and the memory fetch width.

A short instruction word implies at first sight a small space cost. Similarly a short instruction word may imply reduced instruction fetch time, particularly if more than one instruction is packed into one memory word. A slightly shorter decoding time might also result from a short instruction word. However, the advantage of a short instruction word turns into a disadvantage when the set of available features becomes too poor. At some point commonly used operations have to be expressed as a sequence of two or more instructions, and both time cost and space cost rise†. Obviously there is an optimum for both space and time, not necessarily the same, and probably not very well defined††. There is also an associated hardware cost, usually increasing with instruction word size.

To simplify the discussion we will from now on assume that the word length is given, and one and the same for instructions and for integer and real operands. On this assumption we consider the problem of which of the desirable features can be represented within the instruction word. This represents little limitation on the scope of our methods. Data obtained by them are certainly valid arguments in discussions of instruction word size, and the changes in the methods needed to handle more esoteric cases of mixed wordlengths are mostly trivial.

The next issue brought up is the division of the instruction word into fields. Each field represents some <u>capability</u> of the ISP, such as operator selection, addressing mode selection, operand selection etc. Which capabilities to include is an open question, indirect addressing and base register addressing being cases in point.

Having decided which capabilities are wanted, there is the question of the size of each field, and which <u>functions</u> to include for each capability.

Knowing the relative values of the possible functions in a capability and given its field size,

- - - - - - -

† A similar argument holds for data word lengths, in that case it is the need for accuracy which pushes towards longer words.

†† In particular this depends on the application.

one may select a set of functions for it. Some idea of the relative merits of functions from different capabilities is necessary to decide on the field sizes, or on the desirability of having a given capability at all. Note that a function becomes particularly expensive when the field capacity† of that capability is about to be exhausted. This means trading it against a considerable reduction in some other capability or against an increase in the instruction word size. In fact, the cost paid is usually that of doubling†† the number of functions. Once this cost has been paid, however, functions that would not otherwise have been considered, can be implemented cheaply.

The goal of our methods is to estimate the relative costs and usefulness of capabilities and their functions. They thus give exactly the kind of information that sheds light on the problems of how to allocate the instruction word space to capabilities and functions.

The allocation of functions to capabilities is not unique. Also structural changes in one capability may imply significant changes in another. One example is provided by two address ISPs. When both operands can be accessed by a full address, the traditional LOAD and STORE instructions are subsumed by a MOVE instruction. Another example is the handling of I/O devices. Commonly there are instructions like "connect", "send function" and "read status" to control these. On the PDP-11 this is not so. The relevant registers of the external devices have been allocated functions in the addressing capability and the above instructions are subsumed under the MOVE instruction. Yet another example is provided by general registers. If these are part of the addressing space, register to register functions are not needed in the operation capability, they are subsumed under the memory to register functions.

2.2 Time cost

The primary time cost is the time the central processor spends executing the program. For reasons explained in Section 1.3 the primary time cost excludes time spent in interrupt handling, whether the program's own or others'. Unless specifically mentioned, the term time cost is used to mean primary time cost.

- - - - - - -

† Usually some power of two.

†† Assuming a binary instruction word.

Execution time can not be measured directly by our methods. We propose three approximations:

One is the instruction count, i.e. the number of instructions executed. This suffers from the inaccuracy caused by assuming that all instructions execute in the same time. This is further discussed below. Modifications could be made depending on addresing mode (particularly indirection) and other features. This was not done in our case. The major advantage of this measure is the ease with which it is computed, and its independence of technology† and processor implementation. The instruction count also has another quality: In addition to being a crude measure of time, it is a precise measure of the number of opportunities there have been to express something in the program.

For many designs, the memory reference count may be more appropriate. The PDP-11 is a good example of this, since for the same data operation the number of memory accesses varies depending on addressing mode. In case of the ADD instruction the number of memory accesses may thus vary between 1 and 7.

If there is no overlapping between instruction executions, a more accurate measure is the computed time, that is the sum of the execution times of all instructions executed. Even this is inaccurate since execution times of many instructions depend on operand values or lengths and also on hardware, like primary memory cycle time. The latter may vary even within the same run if the job is swapped. However, the time obtained in this way is probably as accurate as that used for accounting and other purposes by operating systems, where operating system overhead and interrupt handling on behalf of other jobs often is a major source of errors.

We may get an indication of the inaccuracy of the instruction count as a measure of the time cost by comparing it with the computed time. This is done in Figure 3-4, which displays the average instruction execution rate for our subject set in units of thousand instructions per second of computed time (kips = kilo instructions per second). As the table shows, this rate varies from 210 to 417 kips, with an average of 324 kips and a standard deviation of 63. Hence the instruction count may vary by a factor of 2 for programs of the same

----

† A faster floating point unit would make a great difference in the execution time for many programs, but not in the instruction count. In one of our subject programs (Aitken E, see Section 3.2.2), 23% of the executed instructions, consuming 54% of the computed time, are for floating point arithmetic.

computed time. Assuming the computed time to be close to correct, we may conclude that the instruction count is not overly accurate as a measure of time. We still use it, however, for the stated reasons.

For a central processor where there is overlap between instruction executions the instruction count may be sufficient. Alternatively an interpreter for the instruction dispatching mechanism may be programmed and an appropriate version of computed time obtained. The choice depends on whether one wants to evaluate the instruction set as such, or the processor that executes it. Such an interpreter might introduce additional inaccuracies.

The relations between the time and space costs through the instruction word are described in Section 2.1. The tradeoff discussed there applies to all capabilities and functions of the instruction word, and also to the implied data types.

The secondary time cost is the time spent in operating systems functions on behalf of the running job. This can be measured by clock or by using operating system routines as the subject programs of the analysis. This cost is influenced by the space cost as discussed in Section 2.3.

2.3 Space cost

This is the cost of the primary memory that a program occupies for code and data (static and dynamic). The importance of this cost follows from the relatively high cost of primary memory, which is commonly an expensive part of a computer installation†.

Contributing to the space cost is instruction space and data space. Given an application both of these will vary with the ISP, in particular with the available data types and their operators. Variations in register structure and control operators will influence program space and space for temporary storage.

----

† With the current trend towards semiconductor memories, the technology is the same for the memory and the processor. Since the memory is usually much larger (in gates), memory cost will continue to be high until another technology becomes economical.

Space cost is best measured by static methods or by estimation based on miscellaneous assumptions as relevant in the particular case. The data space for dynamic data structures can not be measured by static means. It can be measured by dynamic methods, but we present no method for this at the present time.

For static methods one may rely on the compiler in question to produce the statistics, or a special program may analyze core images, relocatable programs or some similar general form of the program. The first approach suffers from lack of generality as discussed in Section 1.2.1. The second may have inaccuracies due to the difficulty of distinguishing instruction words from data, in particular constants and descriptors. This inaccuracy depends on the central processor structure, it will be small or nonexistent on a central processor where code and data are completely separated, as on the HP 3000.

Space cost is measured in bits, alternatively in words. Whenever we estimate this cost there will be inaccuracies inherent in the particular assumptions made. These will be discussed in each case.

Memory access width relates the space and time costs by forcing unnecessary space to be used rather than increasing the time cost. Memory access width is again influenced by the amount of space necessary for representing data types. Dynamic methods may be desirable here, to determine the space necessary to represent the actual significance of numerical operands (See Section 5.5).

Also space cost relates to time cost through the instruction word as discussed in Section 2.1. For a computer with a dynamic memory management (paging, overlaying) there will be an associated secondary time cost for this function which usually increases with the space cost. In a multiprogrammed situation there will also be a relation to secondary time cost through central processor idle time whenever the program is difficult to multiprogram. This also increases with the space cost.

2.4 Programming cost

This cost may be broken down as cost of design and coding, debugging and maintenance. Costs incurred by errors during production runs may also be included. Each of these is often a significant fraction of the costs associated with a program. The most important way of reducing the programming cost is to write programs in high level languages. However, for efficiency reasons, and in order to gain access to machine features, much coding still takes place in assembly languages. Similarly most debugging is done by means of assembler oriented debuggers, or at least requires good knowledge of the representation of the program in ISP terms. Hence a good ISP architecture contributes to reducing this cost in several ways:

By supporting high level languages and other good programming methodologies. This includes techniques for program factorization, like subroutines, coroutines and separately compiled modules, which should be well supported by the ISP. Also important are natural representations for a rich set of other control operators and their associated data structures.

By supporting program security. A program should be protected against its own errors as well as those of other programs. The instruction set should not encourage the programmer to make unnecessary mistakes, and the ISP should permit inconsistencies to be detected during execution†. Possible dynamic checks could be: consistency of data types and operators, validity of effective address with respect to named data structure, consistency of control operators and their data etc. The standard techniques for protection against other programs are to a lesser extent relevant to our subject.

By having the right operators. That is: fewest possible operators should have to be fabricated from existing ones. This contributes to understandability. For particular languages or application areas instructions for indexing in two dimensions, parameter checking, etc. might be relevant.

By being clean and elegant. This means that the capabilities and their functions should be well defined and conceptually well separated (orthogonal). There should be few and well defined instruction word formats. The data types and control operators should be well defined, and their representations should be easily understandable. General concepts should be preferred to special.

The methodology and elegance dimensions of this cost are currently not quantifiable except by purely subjective evaluation. Personal biases and preferences will have a strong

- - - - - - -

† Wirth, [WirN72] has stated the case for this form of security and its dependence upon the ISP very eloquently. See Section 1.4.

influence. As for the security dimension, the cost and value of proposed checking mechanisms can be estimated using our methods to obtain data on dynamic usage. We also provide methods for evaluating existing and missing operators, namely the frequency counts and FGR function (Section 5.1 through Section 5.1) and the sequences (Section 5.2).

Except for the "right operators" dimension, most of the programming cost is accumulated over features missing from the ISP. Introduction of new features, to lower the programming cost, will usually be at increased space, time and hardware costs. However, a generalization of existing features will often entail a reduction of all costs.

We have discussed this cost partly to point out that security measures can be built into the ISP at some (often low) cost in space and time, and that our methods can be used to estimate these costs. We also want to point out that we do not advocate rushing headlong into making some improvement suggested by our methods to save space or time, without considering the issues just discussed.

2.5 Hardware cost

This is the cost of the hardware of the central processor needed to implement a feature. Given the approximate computing power of the processor and its general structure, the varying part is mostly a cost of electronic circuitry. Since the cost of integrated circuits is rapidly falling and becoming a small fraction of the cost of a computer system, the hardware cost is becoming less significant.

Estimating the hardware cost is outside the scope of this thesis. As a general rule each feature introduced into the ISP will increase it, less so if the new feature, or part of it, is subsumed under an already existing concept and using existing hardware. It follows that an increased hardware cost is usually the consequence of an improvement designed to reduce the space and time costs.

Time cost can be reduced by using faster circuits, thus increasing the hardware cost. This is irrelevant to the ISP architecture. Hardware cost is independent of space cost, its relation to programming cost is discussed in Section 2.4.

CHAPTER 3

VALIDATION STRATEGY

A major concern of our research has been to establish the validity of the methods we have developed. We wanted to ascertain that they apply with more or less equal generality to the ISP structures outlined in Section 1.3 and to all application areas where this class of processors is commonly used. We wanted to be confident that the results obtained by using them reflect general requirements of programmers, algorithms, languages and compilers rather than idiosyncrasies of particular instances of such. Specifically we wanted to assess the influence of each source of variation on our results.

The sources of variation can be grouped as:

- Variation due to algorithm.
- Variation due to programmer.
- Variation due to language used.
- Variation due to the particular implementation of that language (including the operating system).
- Variation due to the ISP.

One might also want to consider variation due to choice of representations, particularly for data structures. This variation is closely related to those due to algorithm, programmer and language, and we do not treat it as a separate source of variation here.

The validity of the results have been judged by several criteria:

The methods confirm already known efficiencies or deficiencies of the ISP considered.

The methods give new insight into deficiencies or efficiencies of the ISP which are subsequently verified by other means.

The methods themself may measure or illuminate the same property of the the ISP from several angles and these results corroborate each other.

In special cases the approximate measures found can be compared against direct measurements.

In this chapter we describe some simplifying assumptions which were made, and how we chose a subject set in order to investigate the influence of the above sources of variation. As the presentation of each method, and the experimental results obtained by it, is concluded, we also discuss the results in view of this validation strategy. Finally these discussions are summarized in Section 8.2.

3.1 Some simplifying assumptions

To make a full scale investigation of the effects of all these sources of variations would be a major programming task. Particularly costly is tracing on several ISPs, and selecting the subject programs from a wide area of applications. Firstly we would need an interpreter program for each of the ISPs to be investigated. Secondly, we would have to change the analysis programs to reflect the other ISPs†. Thirdly, in selecting subject programs we would need several programs from each major area of application. These would have to be coded in each of the selected languages and brought to run on each of the selected ISPs before analysis of them could start. The analysis would entail a large expense in computer resources and the result would bring on us a data reduction problem of considerable magnitude. In addition it would involve locating and consulting experts in each application area.

We believe that we have legitimately evaluated our methods without going to this large scale investigation, by introducing two simplifying assumptions:

1) We restricted ourselves to one ISP, viz. the PDP-10. This alleviated the first two difficulties above, but deprived us of the possibility of investigating the variation due to a change of ISP. Almost all of our experimental results would change if we performed our analyses on a different ISP, particuarly the results for register utilization, details of instruction sequences, and addressing. In some cases the

----

† There is an obvious advantage of running the analysis programs on the same processor as is traced, since many of the representations have obvious and efficient formats. Most of our programs were written in FORTRAN to ease portability, but even so many of the representations would have to be changed when tooling for another ISP.

methods would have to be modified, or new methods developed, to handle special features of particular ISPs†. We believe this to be of little importance in the present context. Our goal was to assess the ability of our methods to detect the utilities and costs of features in ISPs, as opposed to comparing ISPs. Since our methods justified themselves for one ISP we feel confident they will work satisfactorily for most. Analogously, if we were developing methods to determine the cost/utility ratio of programming language features based on their usage, we would certainly measure the performance of programs on several ISPs but we might well restrict ourselves to one language provided it were sufficiently rich. Further justification follows from the generality of the PDP-10 as discussed on page 9. If the findings of our validation did not have a certain generality to them we would suspect this assumption of failing. As it is, we don't.

2) We restricted ourselves to one, albeit rather general, area of application. This reduced the set of subject programs to manageable proportions. Again, we believe that since our methods showed their worth in evaluating an ISP over one application area then they can be applied over a spectrum of areas, separately or in union. We would expect the findings to differ from area to area but mostly in data types and data operators. This is probably the best understood part of the domain that our methods can be applied to and hence of least importance to us. We would also expect data accessing methods to be influenced by the application and our assumption deprived us of assessing this influence. Considering this assumption, we restricted our study to programs mostly from the area of technical and scientific computations, but with some other programs included, in particular compilers.

We summarize this discussion as follows: The intended goal of our methods is to evaluate features of ISPs as suitable for a given general or specialized application area. Our main concern in validating the methods was to assess the influence of factors not related to the ISP or to the area of application.

----

† Consider the IBM 360 ISP as an example, and compare it with the PDP-10. Base register addressing would imply that more registers would be used, and that information about addressing would become more important. The differences in instruction sets would imply changes, at least in detail, of the instruction sequences. Also methods for investigation of the use of condition codes would have to be implemented.

3.2 Selection of data

Again, since we evaluated the methods, and not any particular ISP, we were not worried that our selection of subject programs quantitatively constituted a fair representation of any actual workload. Rather we wanted to see all programming structures that occur with some minimal frequency in real world programs represented in our test sample. To estimate the influence of the various sources of variation we studied the behaviour of several versions of the same several algorithms, programmed by different programmers, in different languages and, if possible, compiled by different compilers for the same language.

3.2.1 Language selection

To study the language variation, we selected four available languages suited to the chosen application area, namely: FORTRAN, ALGOL, BASIC† and BLISS. These languages cover a range of age, degree of security, inherent efficiency and structure:

FORTRAN [IBM56], [USAS66] was designed about 1954 but has since been modified and extended considerably. ALGOL [NauP63] was designed in 1957-60, BASIC [KemJ61] in the early sixties [KemJ61], BLISS [WulW70] was designed around 1969.

In terms of control structures, including program factorization mechanisms, all the chosen languages have looping and conditional constructs. BASIC is the poorest, having subroutines but no local names. FORTRAN has more structure, particularly subroutines and localized data. ALGOL has even more, notably the compound statement with its consequences for the other control structures, block structure, and an advanced parameter mechanism. BLISS is comparable to ALGOL, with a simpler parameter mechanism, but it has coroutines, and intra routine control structures so rich that a general GO TO has been omitted. This contributes towards better structured programs.

For data structures, FORTRAN, BASIC and ALGOL all have vectors and multidimensional arrays, BLISS has any data structure which the programmer cares to define.

- - - - - - -

† To obtain a fair comparison of the language structures involved, we did not use the matrix operators of BASIC where they would normally be called for.

BASIC has only one type†, floating point, converting to integer indexes automatically as needed. ALGOL and FORTRAN have several arithmetic types with automatic type conversion, and also a Boolean type. BLISS has no types but relies on the written operator to determine the correct operation.

FORTRAN and BLISS have almost no run time checking, BASIC checks array bounds, ALGOL does this and also has extensive checking of parameters including type conversion.

BLISS generates the most efficient object programs, largely due to a highly optimizing compiler. FORTRAN programs are efficient, ALGOL programs are less efficient due to the high degree of security and to the precise definition of evaluation order in the context of possible side effects. BASIC programs are inefficient due to a particularly fast and dirty compiler.

It follows that our languages span most of the variations found within commonly used languages for scientific and technical calculations.

3.2.2 The subject set

For our subject programs we first selected six algorithms from the "Collected Algorithms from the Communications of the ACM", (CALGO). The selection was made in such a way that it included as many as possible of the common data types, data structures, control structures and parameter forms found in higher level languages. We also attempted to cover as wide a range as feasible of the modified SHARE classification, used by CALGO to classify the algorithms. Other criteria used in the selection were:

> The algorithm must have a reasonable size, - large enough to contain the interesting features in context, but small enough to be coded in all four languages, traced and analyzed in a reasonable time.
>
> The remarks and certifications in the CALGO collection should not indicate that trouble might be expected using the algorithm.
>
> The subject matter of the algorithm should be sufficiently known to this author that he could detect obvious errors in the published algorithm and in his various versions of it.

- - - - - - -

† Excluding the string type which we don't use.

Writing a main program for the algorithm should be straight forward.

The CALGO algorithms selected are briefly described in Figure 3-1, along with the rest of the subject set. This set of algorithms gives us a good indication of the variations due to algorithm and language. Listings of all the ALGOL versions, all 4 versions of PERT, and all 5 versions of Aitken, are reproduced in Appendix E.

The language structures searched for, showing how they occur in the selected algorithms, are tabulated in Figure 3-2. The statement count given is the approximate number of ALGOL statements† in the published version, included as a measure of the coding effort. As is seen from the table, several of the desired structures are not represented. Double precision arithmetic is only present in one algorithm, Crout, very locally in space (though not in time), and only in the ALGOL and FORTRAN versions since BLISS and BASIC do not support this type. Complex arithmetic is only marginally present, since Bairstows method finds complex roots but does no calculations using them and no variables are declared of this type. Bit manipulation, bit vectors and characters are not used by any of these algorithms. Note also that real arithmetic in treesort is present only to the extent in which it is needed for comparisons of magnitude, or for initialization.

Only Crout's method uses two dimensional arrays and we found no suitable algorithm using arrays of 3 or more dimensions††, and no triangular or ragged arrays. We also found no suitable algorithms using record structures or lists, although Treesort uses linked structures.

We found a rich selection of GO TOs†††, conditionals and loops, and one instance of a CASE statement (switch, computed GO TO). Since only BLISS and ALGOL support recursion, and this feature is little used in published algorithms, we did not include it. For the same reason we included no algorithm using label parameters. Other parameter forms are well represented. In particular, Ising passes procedure names as parameters. For this reason Ising could not be coded in BASIC.

- - - - - - -

† Not counting <block>s and <compound statement>s. Thus "IF B THEN BEGIN A:=X+1; I:=I-1 END ELSE A:=X-1;" counts as 4 statements.

†† Knuth [KnuD70] reports that 1.4% of the static variable occurrences in his FORTRAN sample has 3 or 4 indices or parameters. He does not distinguish function calls from array accesses. Assuming functions of many parameters to be more common than arrays of many dimensions, this supports our findings.

††† Most of the GO TOs caused little problem when translating into BLISS, an exception was the Bairstow program which required artificial loops, compounds and a function.

FIGURE 3-1

Description of the subject set.

CALGO no. 30 Bairstow	Bairstow/Newton method for polynomial roots. Author: K. W. Ellenberger. Corrections by W. J. Alexander, K. J. Cohen and J. J. Kohfeld. Modified SHARE category C2: Zeroes of polynomials. Data: Initialization by explicit assignments. This is a classical algorithm for the problem.
CALGO no. 43 Crout	Crout's method for linear equations with pivoting. Author: H. C. Thacher. Corrections by C. Domingo and F. Roderiguez-Gil. Modified SHARE category F4: Linear equations. Data: Matrix values computed by simple expressions. Logarithm used for right hand sides. A classical algorithm for the problem.
CALGO no. 113 Treesort	Treesort. Author: R. W. Floyd. Modified SHARE category M1: Sorting. Data: Initialization by simple expression. Initial order is inverse of desired. A logarithmic sorting algorithm.
CALGO no. 119 PERT	Evaluation of a PERT network. Authors: B. Eisenman and M. Shapiro. Corrections by L. S. Coles. Modified SHARE category H: Operations research, graphs. Data: Initialization by explicit assignments. A somewhat speeded up algorithm for this problem.
CALGO no. 257 Håvie	Numerical integration by Håvies method. Author: R. N. Kubick. Modified SHARE category D1: Quadrature. Data: Integrands are simple expressions involving square root or exponential. A modified Romberg integration.
CALGO no. 355 Ising	An algorithm for generating Ising configurations. Author: J. M. S. Simoes Pereira. Modified SHARE category Z: All others. Data: Maximal n read from teletype; n, x and t varied by loops over all significantly different combinations. An (x,t) Ising configuration is a sequence $(S_1,\ldots,S_n)$ of zeroes and ones such that: $\sum_{i=1}^{n} S_i = x$ and $\sum_{i=1}^{n-1} \lvert S_{i+1} - S_i \rvert = t$ The problem is of interest in theoretical physics.

This algorithm was included mainly because routine calls is its most important control structure. Since routine names are passed as parameters it could not be coded in BASIC.

Aitken	N-point polynomial interpolation. Authors: M. R. Barbacci, L. E. Flon, G. N. J. Rolf, W. A. Wulf and A. Lunde. (Each contributed one version of the algorithm. The slowest version was omitted. The fastest (and shortest) version was further improved by about 10% in speed and size, and included. Hence five versions of this algorithm were used.) Modified SHARE category E1: Interpolation. Source language: BLISS. Data: Natural logarithm tabulated at irregular intervals by loop. Standard polynomial interpolation.
SEC	Zeroes of simultaneous nonlinear equations by secant method. Author: G. W. Stewart. Modified SHARE category C5: Zeroes of trancedental functions. Source language: FORTRAN Data: Functions are linear combinations of linear and quadratic terms in the variables, parameters read from teletype. The program was designed for research in the problem area and method.
FORFOR	Compiler for FORTRAN. Source language: Assembler. Data: FORTRAN version of the Treesort algorithm. A compiler of the Digitek design, simulating a one-accumulator processor.
FORTEN	Compiler for FORTRAN. Source language: BLISS. Data: FORTRAN version of the Treesort algorithm. A compiler doing flow analysis and generating efficient code.
ALGOL	Compiler for ALGOL. Source language: Assembler, structured control by macros. Data: ALGOL version of the Treesort algorithm. A fast ALGOL compiler generating efficient code (for ALGOL). Language slightly extended.
BASIC	Compile and link phases of the BASIC system. Source language: Assembler. Data: BASIC version of the Treesort algorithm. A fast compiler generating extremely inefficient code.
BLISS	Compiler for BLISS Source language: BLISS. Data: BLISS version of the Treesort algorithm. A slow compiler generating efficient and small code.

FIGURE 3-2

Language properties of the small subject algorithms:
x means property present in algorithm.
- means property marginally present in algorithm.

Name:	Bairst.	Crout	T.sort	PERT	Håvie	Ising	Aitken
CALGO number:	30	43	113	119	257	355	-
Mod. SHARE categ.:	C2	F4	M1	H	D1	Z	E
Statement count:	120	40	15	60	35	45	30
<u>Types:</u>							
Integer	x	x	x	x	x	x	x
Floating	x	x	-	x	x		x
Double fl.		-					
Complex	-						
Boolean		x					
Bits							
Characters							
<u>Data structures:</u>							
1 Dim arrays	x	x	x	x	x	x	x
2 Dim. arrays		x					
>2 Dim. arrays							
Ragged/triang. arr.							
Records							
Lists							
Linked			x				
Packed			x				
<u>Control structures:</u>							
Go to	x	x		x	x	x	
Conditionals	x	x	x	x	x	x	x
Cases				x			
Counting loops	x	x	x	x	x	x	x
Other loops	x		x		x	x	x
Subroutines		x		x	x	x	
Recursion							
<u>Parameter forms:</u>							
Constants		x				x	
Variables		x		x	x	x	
Expressions		x				x	
Arrays				x		x	
Routines					x	x	
Labels							

A related source of variation is that of language implementation. Luckily the PDP-10 has two FORTRAN systems, FORTRAN 40 and FORTRAN TEN, here denoted FORFOR and FORTEN or simply FOR and TEN. Hence we had an obvious way of assessing this variation. We analyzed all the CALGO algorithms plus SEC (see below) using both of the FORTRAN systems. Due to a suspected bug in TEN, we did not use the optimize option of TEN when compiling our programs. The various versions of these algorithms will be denoted ALGOL Ising, BASIC Crout etc.

To estimate the variations due to programmer habits we included 5 versions of an algorithm as coded in BLISS by 4 experienced programmers. The algorithm was polynomial interpolation† which nicely completed our coverage of the modified SHARE categories. BLISS was chosen since it gives the programmer more alternative forms of expression than do the other languages. This was thought to be of importance considering the small algorithm. These five programs are denoted by the letters L, G, B, A and E (efficient).

For each of these algorithms a main program was written, to provide data for the algorithm and present the results. To initialize the data for the algorithms we used explicit assignments of either constants or calculated values, usually simple expressions involving the indices of the variables to be initialized. A short indication of the method used in each case is given with the description of the algorithm in Figure 3-1.

After a few trial traces it became obvious that input and output accounted for a large fraction of the total activity. Not only did format interpretation take much time, but also channel and file initialization and status checking. We therefore decided to leave I/O out of the traced part of the algorithms, with a few exceptions: one parameter to the Ising program is read from the teletype, and a minimal output was included in some cases.

Our sample so far had one major deficiency: all the programs traced were small. To rectify this we traced all the compilers involved, that is the ALGOL and BLISS compilers, the compile and link phases of the BASIC system and the two FORTRAN compilers. All these traces were made while compiling the appropriate version of the Treesort algorithm. An additional benefit from this was that we got examples of many of the structures our CALGO sample did not have, including bit manipulation, bit vectors, character handling, records, lists and

----

† By Aitkens method as described in Milne [MilW49].

recursion. We also believe that compilers account for a large fraction of the resources used in any installation and hence are of particular importance as constituents of sets of typical programs.

We further included one somewhat larger program from the technical scientific calculations area, this was a program, SEC, to solve nonlinear simultaneous equations. This program was analyzed using both versions of FORTRAN.

The resulting subject set consists of the 6 CALGO algorithms written in each of the 4 languages, the Aitken algorithm written in BLISS by 4 programmers, 5 compilers and the large scientific numerical program. These programs are well distributed over the area spanne. by the modified SHARE classification. The following general categories are represented:

B (Standard functions) by the integrands for Håvie.
C (Polynomials, zeroes) by Bairstow and SEC.
D (Integrals and differential equations) by Håvie
E (Polynomial approximation) by Aitken.
F (Matrix operations) by Crout.
G (Statistics, permutations, subset generation) by Ising (related).
H (Operations research, graphs) by PERT.
L (Compiling) by the compilers.
M (Sorting, data conversion) by Treesort.
Z (Others) by Ising.

The FORTRAN versions of the 6 CALGO algorithms, and also the large scientific program, were analyzed as compiled using the two different FORTRAN compilers. Thus, since the BASIC version of Ising was excluded, the sample altogether consisted of 41 traces. The traces vary in size from 19000 to almost 600000 executed instructions. Altogether about 5.3 million instructions were traced, corresponding to almost 16.8 seconds of CPU time (***computed time***) on the KA10. This should give a good basis on which to evaluate the methods. The computed time and instruction count of the subject set are tabulated in Figure 3-3. The average instruction execution rate for each program is tabulated in Figure 3-4.

FIGURE 3-3

Time cost of the subject set.
Computed time in seconds.
Instruction count in 1000s.

Source language:	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.12	0.45	0.09	0.08	0.08
	36	156	23	21	19
Crout	0.32	0.49	0.25	0.43	0.23
	115	163	62	109	63
Treesort	0.47	0.55	0.26	0.27	0.35
	140	187	106	111	97
PERT	0.16	0.41	0.07	0.08	0.07
	63	157	26	32	27
Håvie	0.48	0.33	0.12	0.18	0.17
	168	103	28	38	36
Ising	0.22	-	0.07	0.05	0.05
	91	-	25	20	20
SEC	-	-	-	2.08	1.94
	-	-	-	541	497

Algorithm\Programmer	E	B	A	G	L
Aitken	0.18	0.19	0.21	0.41	0.44
	44	47	60	143	139

	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Assembler written	0.19	0.25	-	1.56	-
compilers	74	85	-	591	-
BLISS written	-	-	1.67	-	0.78
compilers	-	-	593	-	295

BLISS versions would have been faster if OWN vectors and matrices had been used instead of LOCAL and parameter.

WARNING: The format of this table is slightly different from the standard table format of the later chapters, first used in Figure 3-4.

FIGURE 3-4

Instruction execution rate of the subject set
in units of 1000 instructions per second (kips)

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	300	345	261	247	243
Crout	362	330	249	256	277
Treesort	300	339	401	412	275
PERT	394	380	397	395	402
Håvie	351	308	230	210	219
Ising	410	-	379	391	417
Secant	-	-	-	260	256

Algorithm\Programmer	E	B	A	G	L
Aitken	245	243	282	344	318

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	382	343	354	379	379

Max: 410, Min: 210, Average: 324, Standard dev.: 63.

3.2.3 Subsets of the subject set

In some cases it is desirable to study the experimental results from a subject set representing a subarea of the area of application. Our subject set falls naturally into three such subsets:

a) The compilers.
b) The numeric set consisting of SEC, Bairstow, Crout, Håvie and Aitken.
c) The nonnumeric set, consisting of Treesort, PERT and Ising.

This subdivision is used in Section 5.1.

CHAPTER 4

REGISTER STRUCTURE

We will now discuss the motivation for, and costs associated with general register designs. The main problems we attack are:

a) What is the optimal number of registers? This is the most important issue in connection with register structure. All the costs discussed below depend heavily on this number.

b) How desirable is generality? This can be an issue in some cases, particularly for designs with a short instruction word.

We do not pretend to solve these problems, only to present methods for elucidating them.

The central concept in our methods is that of a <u>register life</u>. We present an algorithm for detecting such lives, a method of classifying them according to the types of the events constituting them, an algorithm to detect simultaneous lives, and finally methods to estimate the cost of simulating parallel register activity in fewer registers than were used by the original subject program as traced. The data obtained by these methods are highly relevant to the problems of register block size and generality. The first few subsections discuss register structures in general, terminology, and other topics common to the methods.

4.1 The basic tradeoffs

In old ISP designs, the arithmetic registers that the programmer had access to were the actual input registers to the arithmetic unit. A typical design would have an accumulator (A register), and an extension of it (Q register) to hold double length products and dividends, quotients, multipliers, and the like. The second operand for arithmetic would come from primary memory. Further there would be a number of index registers which would have a restricted set of arithmetic and testing operations. From a slightly different viewpoint one might say that the registers were divided into groups according to criteria such as:

Floating point capability
Full fixed point capability
Simple fixed point capabilities and indexing
Temporary storage only
etc.

The "simple fixed point" group could be those having addition and subtraction only, possibly further restricted to immediate operands only.

As electronic circuitry became cheaper and faster compared to primary memory it became feasible and common to have a small electronic memory in the central processor for locally important operands. Operands, as specified by an extra address in the instructions, are transferred through a switch from these memory cells to the arithmetic input registers, whereas the latter registers are invisible to the programmer. One or both of the operands may come from this memory, the alternative being primary memory as before. As a natural extension, this memory contains not only the arithmetic operands but also the indexes, control information etc. The terms registers, register block, and in particular general registers, are now used to mean this local memory.

The general registers commonly serve a combination of several functions:

Arithmetic registers
Index registers
Base registers (double indexing)
Subroutine linkage
Program flag registers (for Booleans)
Stack pointers
Address pointers (to data)
Temporary data storage
Temporary program storage (for small loops)
Program counter (PC)
etc.

Few, if any, computers have registers with all these properties. In particular, few machines have the PC in a general register (exception: the PDP-11), and few may execute programs from them (exception: the PDP-10). The register block may be part of the memory address space for all functions (as in the PDP-10), just for some (as in the UNIVAC 1107), or not at all (as in the IBM 360).

We will devote this section mainly to registers for data manipulation. Indexing and indirection will be discussed, however, to the extent that they are operations involving registers.

Assuming that indices, if they exist at all, are always held in "registers" addressable by short addresses in the instruction word, we may list several factors that motivate the transition to a general register design:

> To save addressing space in the instruction word compared to two address designs. This is not discussed further in the thesis.
>
> To save code space and instruction excutions compared to single accumulator designs. To estimate this factor is outside the scope of the thesis.
>
> To have a fast store for locally important operands. This is further discussed in Section 4.6
>
> To have a full complement of operators for indices and control information as well as for normal arithmetic operands. We discuss this in Section 4.5.
>
> To clean up the ISP [illegible]tecture and central processor design. This is again motivated by programming [illegible]d hardware considerations, to estimate its cost and utility is outside the scope of this thesis.

The costs of general registers are contributed by:

> Space cost of lengthened instruction words compared to one address design. This question is not addressed in the thesis.
>
> Time cost of load and store instructions compared to a full two address design. Some of the results of Chapter 5 may bear on this factor.
>
> Time cost of saving and restoring registers. This can be reduced by having special "process swap" or "register save/restore" instructions, or by having separate blocks of registers for each program or for groups of programs, commonly defined by the interrupt structure. Hence this cost may or may not apply on interrupts. The cost certainly

applies on subprogram calls, particularly if subprograms are separately compiled†. Again some of the results from Chapter 5 apply.

Time cost of register access switch. This time is small compared to the time gained by not accessing primary memory, but may increase somewhat with the number of registers. It may be estimated from the results in this chapter.

Hardware cost of the registers and the switch. To estimate this is outside the scope of the thesis.

The relative importance of these factors depends on the state of technology. In particular the current trends towards cache memories, and towards larger, faster and cheaper electronic memories, tend to make the fast local store argument less important. To make valid design decisions when faced with cost effectiveness requirements, it is necessary first to establish quantitatively their relative importance in a technology independent way.

4.2 Some definitions

The intent of these definitions is to make precise the term "register life", and to define some important properties of register lives.

- - - - - - -

† Our analysis of the trace of the BLISS compiler indicates that a "declarable register" is restored more than 5000 times every second due to subroutine calling; the same number as by restoring 16 registers 312 times. A complete process swap would thus have to be performed over 300 times per second in order for the time cost of register saving due to process swaps to exceed that due to subroutine calling. We believe this is a high frequency of process swaps for the PDP-10 (KA10), but not extremely high. Including the "F-register", the count for BLISS rises to 16500 registers per second, corresponding to about 1000 process swaps per second. (This is about 1.15 registers saved per routine call). The "temporary registers" are not included at all in these counts. Measurements performed on the IBM 360/91 indicate about 470 SVCs and I/O interrupts per second. Assuming the 360/91 to be ten times as fast as the KA10, this corresponds to about 50 process swaps per second on the KA10. All this indicates that register saving because of routine calls is significantly more costly than register saving due to process swaps.

A register is *loaded* when a new value is brought into it that is unrelated to its previous value (except for possible use of the old value in the address calculation).

A register is *modified* when a new value is brought into it which is the result of an operation involving the old value as one of its operands.

A register is *used* when it is loaded, modified, employed in address calculation, used as an operand, stored, tested or otherwise referenced from an instruction.

A register is *read* when it is used but not modified or loaded.

Since our finest grain of time is that of one instruction, a register may be loaded and otherwise used at the same time. In a finer time scale this would not be so. Hence we regard the sets of loadings, modifications and readings of a register as disjoint. Their union is the set of all usages of that register. Two other subsets are often needed:

A register is *changed* when it is modified or loaded, it is *accessed* when it is read or modified.

A *register life* (*R-life*) for a given register is the span of time starting when the register is loaded and ending with the last access before the next time it is loaded†. If a register is used in the address calculation of a *load* to itself, this use is regarded as an access in the life prior to the loading.

Typically a register life starts with a LOAD; operations like ADD, STORE, SHIFT etc. may reference the register and possibly modify it during its life, it may be used as a stackpointer, indirect address etc.

The initial loading usage in a register life is called its *first use*, the term *last use* has an equally obvious definition. The first and last uses of an R-life constitute its *transitions*. The *length* of an R-life is the time from its first use to its last use, both endpoints included.

- - - - - - -

† An R-life should be thought of as closely related to its register. Formally this could be incorporated into the definition by defining an R-life to be a triple: <Register name, time of load, time of last use>.

A register is live during an R-life for that register. It is dead when it is not live. It is dormant when it is live but has not been used for some long period of time specified in each actual case.

We emphasize that we are observing the dynamic behaviour of programs, hence the observed R-lives are in general different from those that we would observe by a static study of the code between the instructions responsible for the first and last uses, and the usages of a register during its life may involve instructions from quite remote parts of the code.

The following definitions are introduced in order that we may classify R-lives according to the kinds of operations they have been used for. This will be used to assess the need for generality of registers.

A register usage classification is a set of possible modes or attributes, each describing a different way in which a register may be used by an instruction.

A simple classification could be: {<loaded>, <stored>, <used for integer arithmetic>, <used for real arithmetic>, <used otherwise>}. A more complete classification is presented in Section 4.3.

A register usage attribute is a member of a register usage classification. The above classification has 5 attributes: <loaded>, <stored>, etc.

A register usage class is a set of register usage attributes, i.e. a subset of the register usage classification.

When no confusion can arise, the word "register" is usually omitted from the above 3 terms.

Each R-life has a *usage class* associated with it, which is uniquely defined by the (unordered) set of usages of the register during its life. We will usually use the term to denote a class defined in this way.

A register usage classification is in a sense a generalization of the set of instructions and other basic operations of the processor which involve the registers. It may also be thought of as a classification of the instructions of the ISP in terms of how they use registers. Given an opcode and a field of the instruction word which may specify a register, a usage attribute is true or false depending on whether that instruction uses the register specified by that field in that particular mode. This is in fact the way it is represented in our analysis program.

4.3 A register usage classification

In Figure 4-1 and Appendix C we present a ***register usage classification*** for the PDP-10. It is designed to detect the ***loading, modification*** and ***reading*** of registers, as well as the various forms of ***reading*** or ***modification***. This classification was used in our analysis programs to detect and classify R-lives. Although it is designed for a particular ISP, few and obvious modifications would be necessary to use it for any other register oriented ISP†.

This classification grew and generalized as we were working with it. Our experience is that the classification given in Figure 4-1 is satisfactory. It contains three minor improvements over the one we actually used for our analyses. The "Used as operand" and "Immediate fixpoint add or subtract" ***attributes*** were included post hoc. Also, our analysis program did not check for instruction fetches from registers, only for jumps into registers or XCT†† instructions addressing registers. The errors caused by this omission are considered insignificant.

For technical reasons the machine representation of the register usage attributes separate them into two kinds, reference attributes and access attributes. Reference attributes are used to define the three major types of reference, i.e. ***loading, modification*** or ***reading***. They are used by the analysis programs as case selectors, and hence represented as consecutive values. The access attributes are used to accumulate the types of usage of a register during its R-life. They are represented as bit positions in a field, so that they may be easily included into a ***register usage class*** by OR-ing.

Since there are 3 fields in each instruction word of the PDP-10 which may reference a register, the actual description of each instruction consists of 3 sets of attributes, each corresponding to one of these fields and the different ways it may use a register. Further complication follows from the existence of instructions which reference two registers by the "ACC" field, from the special treatment of register 0 by many instructions, and from the

- - - - - - -

† For example, if analyzing the PDP-11, autoincrement might be introduced as an attribute.

†† Execute contents of effective address

FIGURE 4-1

A register usage classification.

Reference attributes:
- Not used
- Loaded
- Modified
- Used but not modified
- Undefined (Monitor communication etc.)

Access attributes:
- Indexing data accesses
- Indexing jumps or executes
- Indexing immediate operands
- Immediate fixpoint add or subtract
- Fixpoint add or subtract w. memory operand
- Fixpoint multiply or divide
- Floating point arithmetic
- Halfword modified
- Byte loaded or stored
- Modified by logical operation
- Modified by shift
- Used as stackpointer
- Used to hold an address (As in Block transfers etc.)
- Tested
- Used for monitor parameter
- Used as byte pointer
- Used as indirect address
- Used as an operand
- Stored
- Executed (XCT'ed* or fetched as an instruction)

"result to memory" mode of many PDP-10 instructions. These complications affect the reference attributes, hence corresponding code has to be built into the analysis program. In Figure 4-1 we described the classification as independent of these complicating matters. The full classification, as we used it, is reproduced in Appendix C.

\- - - - - - -

* I.e. referenced by an execute instruction

4.4 Register life detection

In order to say anything beyond trivialities about register usage, it is necessary to detect the register lives. The following simple algorithm will do this in one scan over the trace. A ***register usage classification*** is needed which includes at least the ***attributes*** "loaded" and "accessed". As the trace is read, the algorithm keeps for each register the times of its most recent ***load*** and ***use***. For each instruction in the trace, all fields that can possibly reference a register have to be examined with this in mind. Whenever the register is ***loaded*** anew, or at the end of analysis, the ***transitions*** of its most recent R-life are the most recent load and use respectively. In our experiments we used the instruction count as our time measure; the computed time could be equally well used.

As each R-life is detected, its length is immediately known. Similarly the number of references to each R-life, the number of memory and register references etc. are easily accumulated by this algorithm.

Distributions of lifelengths and usages per R-life from a typical analysis run are shown in Figure 4-2. Because of the dominance of short lives but with a significant number of long ones, a logarithmic division was used in the table. These results are too voluminous to present in full for all of our subject programs. In Figure 4-3 we tabulate for each subject program what fractions of all the lives are accounted for by lives of lengths at most 7, 15 and 31 instructions. Similarly in Figure 4-4 we tabulate the fractions of all lives that are accounted for by lives with at most 3, 7 or 15 usages.

A summary of other results of this algorithm from analyzing our subject programs is shown in Figure 4-5 through 4-11. All these results were obtained under the assumption that a register was ***dead*** when it had been ***dormant*** for 200 instructions. The reason for this assumption, and a discussion of its consequences, is given in Section 4.6. For the present results it means that a few lives (the exact number is tabulated in Figure 4-26) are considered as two or more, with correspondingly shorter lives and fewer references per life.

This algorithm is critically dependent on the ability to define the "load" and "access" ***usage attributes*** with the intended intuitive meaning. Certain instruction sequences, like HRR, HRL†

- - - - - - -

† These instructions load the right and left halves of a register respectively, leaving the other half unchanged. Alone they were considered ***modifying*** instructions; however, HRRZ etc., which explisitely change the whole register, were considered ***loading***.

FIGURE 4-2

Distributions of lifelengths and usages per R-life
(FORFOR compiling Treesort)

```
     LIFE      NUMBER
    LENGTH    OF LIVES

    1 -     1    27186 |**************
    2 -     3    37627 |*******************
    4 -     7   100480 |**************************************************
    8 -    15    20661 |**********
   16 -    31     6877 |***
   32 -    63     4542 |**
   64 -   127     3298 |**
  128 -   255     1246 |*
  256 -   511      661 |
  512 -  1023      317 |
 1024 -  2047      196 |
 2048 -  4095      105 |
 4096 -  8191       37 |
 8192 - 16383        5 |
16384 - 32767        1 |

        -       203239 |
```

```
    USAGES     NUMBER
    IN LIFE   OF LIVES

    1 -     1    27186 |**************
    2 -     3    97693 |*************************************************
    4 -     7    70482 |***********************************
    8 -    15     5119 |***
   16 -    31     1700 |*
   32 -    63      583 |
   64 -   127      195 |
  128 -   255       86 |
  256 -   511      187 |
  512 -  1023        8 |
 1024 -  2047        0 |
 2048 -  4095        0 |
 4096 -  8191        0 |
 8192 - 16383        0 |
16384 - 32767        0 |

        -       203239 |
```

FIGURE 4-3

Fraction of R-lives of length at most 7,
of length at most 15,
of length at most 31.

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	≤7	0.771	0.560	0.830	0.852	0.824
	≤15	0.920	0.769	0.913	0.915	0.898
	≤31	0.965	0.995	0.966	0.952	0.930
Crout	≤7	0.709	0.631	0.624	0.606	0.636
	≤15	0.875	0.846	0.884	0.857	0.788
	≤31	0.917	0.988	0.943	0.934	0.939
Treesort	≤7	0.906	0.549	0.882	0.902	0.901
	≤15	0.998	0.769	0.999	0.999	0.998
	≤31	0.999	0.999	0.999	0.999	0.998
PERT	≤7	0.816	0.578	0.902	0.952	0.927
	≤15	0.883	0.783	0.961	0.982	0.979
	≤31	0.930	0.999	0.982	0.990	0.983
Håvie	≤7	0.604	0.756	0.585	0.526	0.808
	≤15	0.734	0.956	0.840	0.767	0.846
	≤31	0.806	0.998	0.918	0.989	0.981
Ising	≤7	0.645	-	0.859	0.888	0.822
	≤15	0.808	-	0.908	0.952	0.936
	≤31	0.885	-	0.960	0.992	0.984
Secant	≤7	-	-	-	0.782	0.603
	≤15	-	-	-	0.930	0.970
	≤31	-	-	-	0.979	0.985

Algorithm\Programmer		E	B	A	G	L
Aitken	≤7	0.601	0.631	0.696	0.927	0.820
	≤15	0.794	0.811	0.853	0.943	0.913
	≤31	0.914	0.925	0.941	0.983	0.970

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	≤7	0.771	0.588	0.804	0.813	0.827
	≤15	0.856	0.801	0.923	0.915	0.897
	≤31	0.910	0.869	0.975	0.949	0.950

FIGURE 4-4

Fraction of lives used at most 3 times
used at most 7 times
used at most 15 times

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	≤3	0.819	0.736	0.830	0.670	0.567
	≤7	0.961	0.994	0.913	0.945	0.921
	≤15	0.990	0.999	0.966	0.974	0.970
Crout	≤3	0.743	0.661	0.444	0.702	0.661
	≤7	0.967	0.999	0.934	0.972	0.951
	≤15	0.989	1.000	0.952	0.993	0.993
Treesort	≤3	0.627	0.741	0.732	0.886	0.602
	≤7	0.998	0.984	0.904	1.000	0.999
	≤15	1.000	1.000	1.000	1.000	1.000
PERT	≤3	0.788	0.755	0.831	0.831	0.896
	≤7	0.963	0.999	0.977	0.990	0.984
	≤15	0.981	1.000	0.988	0.994	0.991
Håvie	≤3	0.574	0.731	0.672	0.614	0.563
	≤7	0.910	0.966	0.853	0.776	0.858
	≤15	0.982	0.999	0.994	0.996	0.995
Ising	≤3	0.640	-	0.832	0.755	0.765
	≤7	0.955	-	0.924	0.975	0.958
	≤15	0.986	-	0.966	0.983	0.995
Secant	≤3	-	-	-	0.603	0.520
	≤7	-	-	-	0.970	0.965
	≤15	-	-	-	0.985	0.986

Algorithm\Programmer		E	B	A	G	L
Aitken	≤3	0.618	0.573	0.772	0.913	0.787
	≤7	0.883	0.893	0.912	0.979	0.964
	≤15	0.944	0.944	0.952	0.988	0.976

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	≤3	0.753	0.523	0.842	0.614	0.870
	≤7	0.946	0.800	0.975	0.961	0.970
	≤15	0.989	0.966	0.994	0.986	0.995

FIGURE 4-5

Number of register lives

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	12985	46101	7727	6133	5831
Crout	51087	52978	15871	46308	23515
Treesort	58088	55686	36493	49017	44269
PERT	24324	156974	11264	12769	10387
Håvie	60262	32189	7710	9504	8160
Ising	35919	-	9310	7196	7024
Secant	-	-	-	198167	175569

Algorithm\Programmer	E	B	A	G	L
Aitken	13425	14390	19626	62495	43650

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	21662	16034	220222	203239	108675

The high number of R-lives for the FORFOR and ALGOL versions of Crout, compared to the BLISS version, is probably due to the use of double length arithmetic in those versions. Similarly the high number of register lives for the ALGOL versions of Håvie and Ising is probably due to the large number of procedure and name parameter calls.

FIGURE 4-6

Average lifelength in instructions

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	12.3	12.3	11.2	12.9	12.9
Crout	13.6	11.3	18.2	15.1	15.9
Treesort	6.1	11.9	9.0	4.2	5.8
PERT	10.9	11.4	8.4	5.0	7.9
Håvie	16.6	11.2	13.5	14.3	20.0
Ising	16.5	-	9.7	5.5	9.2
Secant	-	-	-	8.1	9.6

Algorithm\Programmer	E	B	A	G	L
Aitken	14.3	14.7	13.0	8.9	11.9

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	17.4	23.8	9.7	14.9	11.4

FIGURE 4-7

Usages per R-life

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	4.6	3.6	4.6	4.6	4.4
Crout	3.8	3.7	6.6	3.7	3.9
Treesort	3.9	3.5	4.8	2.9	2.9
PERT	4.1	3.4	3.8	3.1	3.2
Håvie	4.4	3.7	5.8	5.4	5.2
Ising	4.0	-	4.5	3.1	3.3
Secant	-	-	-	3.8	3.8

Algorithm\Programmer	E	B	A	G	L
Aitken	5.4	5.5	5.2	3.9	5.2

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	3.7	6.0	3.5	4.1	3.2

FIGURE 4-8

Average number of live registers

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	4.4	3.6	3.8	3.8	4.0
Crout	6.0	3.7	4.7	6.4	6.0
Treesort	2.5	3.5	3.1	1.8	2.7
PERT	4.2	3.6	3.6	2.0	3.0
Håvie	6.0	3.5	3.7	3.6	4.5
Ising	6.5	-	3.6	1.9	3.2
Secant	-	-	-	3.0	3.4

Algorithm\Programmer	E	B	A	G	L
Aitken	4.4	4.5	4.2	3.9	3.7

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	5.1	4.5	3.6	5.1	4.2

Average number of lives is computed as: (sum of lifelengths)/(program length)

FIGURE 4-9

Memory references per instruction

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.61	0.52	0.50	0.62	0.60
Crout	0.44	0.59	0.50	0.55	0.64
Treesort	0.65	0.50	0.51	0.57	0.63
PERT	0.51	0.47	0.53	0.69	0.63
Håvie	0.30	0.45	0.31	0.44	0.35
Ising	0.40	-	0.60	0.67	0.60
Secant	-	-	-	0.60	0.53
Algorithm\Programmer	**E**	**B**	**A**	**G**	**L**
Aitken	0.45	0.48	0.52	0.50	0.53
Source progr.\Compiler	**ALGOL**	**BASIC**	**BLISS**	**FORFOR**	**FORTEN**
Treesort	0.40	0.32	0.45	0.42	0.40

The instruction fetches are not included in the memory reference counts.

FIGURE 4-10

Register references per instruction

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	1.66	1.05	1.58	1.35	1.37
Crout	1.67	1.21	1.67	1.56	1.46
Treesort	1.62	1.04	1.65	1.28	1.32
PERT	1.58	1.05	1.61	1.25	1.22
Håvie	1.57	1.14	1.61	1.36	1.16
Ising	1.58	-	1.66	1.11	1.13
Secant	-	-	-	1.39	1.33
Algorithm\Programmer	**E**	**B**	**A**	**G**	**L**
Aitken	1.66	1.67	1.69	1.69	1.64
Source progr.\Compiler	**ALGOL**	**BASIC**	**BLISS**	**FORFOR**	**FORTEN**
Treesort	1.09	1.13	1.32	1.39	1.17

FIGURE 4-11

Register references per memory reference

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	2.7	2.0	3.2	2.2	2.3
Crout	3.8	2.1	3.3	2.8	2.3
Treesort	2.5	2.1	3.2	2.2	2.1
PERT	3.1	2.2	3.0	1.8	1.9
Håvie	5.2	2.5	5.2	3.1	3.3
Ising	4.0	-	2.8	1.7	1.9
Secant	-	-	-	2.3	2.1
Algorithm\Programmer	**E**	**B**	**A**	**G**	**L**
Aitken	3.7	3.5	3.3	3.4	3.1
Source progr.\Compiler	**ALGOL**	**BASIC**	**BLISS**	**FORFOR**	**FORTEN**
Treesort	2.7	3.5	2.9	3.3	2.9

on the PDP-10 effectively constitute a *load*, but usages of these instructions in other cases do not. As a consequence, some lives may not be properly detected.

A comparison of the results of our sequence program, as described in Section 5.2, with the listing of the ALGOL run time support system, seems to indicate that this source of error may be significant for our ALGOL programs, particularly Crout, Håvie and Ising, which contain many procedure calls and name parameter transmissions. For the compilers traced there are many halfword loads, but no significant pairs of halfword loads, and for the other programs there are no danger signs in our results.

4.4.1 Summary

We summarize these initial results as follows:

Register lives are in general short, less than 32 instructions. Only for 3 of our 41 subject programs are more than 10% of the R-lives 32 instructions or longer, and for 11 of the programs 99% of the lives are shorter than 32 instructions. The average lifelength is less than 24 instructions for all programs, less than 15 for 32 of them and less than 10 instructions for 14 programs. These results vary systematically with the algorithm; PERT and

Treesort have short lives, Håvie has long lives. The BASIC programs form an exception, they all have lifelengths between 11.2 and 12.3 instructions.

The average number of usages per life varies between 3.1 (FORFOR PERT, FORFOR Ising) and 6.6 (BLISS Treesort). Again the results from the BASIC programs vary little with algorithm (3.4 to 3.7), the other results vary more with the algorithm, but not very systematically except for the two FORTRAN versions. These correlate well with each other.

The average number of live registers is less than 7 for all 41 programs, 4 or less for 24 of them. ALGOL programs generally keep more registers live than do programs in the other languages (See footnote on page 74). The results from the BASIC programs again vary little with the algorithm. The correlation between the FORTRAN versions is not as good as for the lifelengths and the usages per life.

The high ratio of register references to memory references suggest that those registers which are live are effectively used for temporary results.

The influence of language and algorithm is not clear. Generally results from the BASIC programs are almost independent of the algorithm, and the ALGOL results often show a consistent trend, but with some variation. In some cases the correlation between the two FORTRAN versions is good. This indicates that the differences found are due to language and not to implementation. Variations due to the programmer are marked, as witnessed by the results from Aitken.

4.5 Register life classification

Specialization of registers may seem irrelevant in view of the current tendency towards general register structures, and the consequent increased generality of ISP and program structure. However, specialization may be of relevance in short wordlength computers, where the addressing space saved by omitting register addresses can be used for more important capabilities.

To assess the utility of a full set of operators for each register we need to know which kinds of operations are performed on a register during its R-life. One way of obtaining this information is to use a finer ***register usage classification*** than the "loaded", "accessed" one sufficient to determine the lives[†], and to extend the life detection algorithm to compute the ***usage class*** for each R-life. That is: at each usage of an R-life the appropriate ***usage attribute*** is included in the ***usage class***. Hence the number of R-lives in each ***usage class*** may be accumulated.

This method for classifying R-lives has two variants. One is to accumulate the ***usage classes*** strictly for one register life. The other is, for binary operations, to let the the ***usage class*** of the result become the union of the classes of the operands. The former is most relevant when we analyze a structure with very general registers to detect unneeded generality, the second variant can be used on an ISP with specialized registers to see the need for a more general structure. Our experimental results were obtained by the former variant.

The information may be tabulated by the register number, allowing us to see for each physical register how it was used. More interesting is to tabulate, for each ***usage class***, statistics on the number of lives in each class, their average length and number of usages. We call this the usage class table or UCT.

None of our analyses showed more than 200 different ***usage classes***. About half of these account for more than 99% of the total number of lives. Hence the UCT forms a very compact database describing the register usage, which can be manipulated or stored for later use at a low cost. A natural format is to store the UCT sorted by the number of lives in the class, or by the sum of the lifelengths represented by the class. Thus we may cheaply ask questions that were not thought of at the time of the original analysis and, in particular, we may study that UCT which is the union of all the UCTs of the individual subject programs. Unfortunately it was not realized until a late stage in our experiments that the UCTs would be small. Hence we have not saved the UCTs from our analyses.

Several forms of output may be obtained from the UCT. A very simpleminded output procedure, which takes ***usage classes*** as its parameters, can be employed to print data pertaining to all classes that are subsets of, supersets of, or other simple combinations of the classes given as parameters. In this way we may obtain statistics on the usage classes a priori thought to be significant. Another procedure may be used to find combinations of ***attributes*** that frequently occur in the same usage class. The result of such an analysis will be an a posteriori classification of the R-lives corresponding to suitable types of more specialized registers.

\- - - - - - -

[†] The one in Section 4.3 is a typical example

In our case, we believed a priori that the classification into floating point accumulators, fixed point accumulators, index registers with simple arithmetic capabilities and temporary storage only, is of such a significance (See page 41). This belief is well founded in history. We display the fraction of lives in each of these arithmetic classes in figures 4-12 through 4-15. Each class is defined by the "strongest" form of arithmetic used in it, floating point being stronger than fixed point multiply and divide, which again is stronger than fixed point add and subtract. R-lives not used for arithmetic may still be used for logical or other operations. These four classes are disjoint. We denote them: Floating, Fixed, Counter and Noari.

Some other classes were also thought to be of interest. The fractions of R-lives that were used only as storage locations are tabulated in Figure 4-16, this class is denoted Temporary. The fractions of R-lives used for indexing (whether for data accessing, jumps or immediate operands) are tabulated in Figure 4-17. This class is not disjoint from the arithmetic classes, and is denoted Indexing.

Yet another classification of interest is the intersection of the indexing class with the arithmetic classes. We have no concise results for these classes, except the printout of statistics for all indexing classes discussed below.

An output procedure as described above was programmed to print the number of lives, fraction of total number of lives, average lifelength and an interpretation of the *usage class* encoding, for the selected set of classes. It was used to print the whole of the UCT as well as the subclasses for arithmetic and indexing discussed above. An example of this output is given in Appendix B.

A study of these printouts brought up several questions which could not be quantitatively investigated since we did not have access to the old UCTs. We formulated several hypotheses, however, and checked them manually in a scan over all the printed results.

1) A significant number of lives are of length one. This was verified. Some partial explanations could be: Values of subroutines returned in registers but not used at the call site. Double length results of integer multiplication and two results of division (quotient and remainder) where only one is used. Linenumbers of BASIC programs are loaded into a register for each source line executed, these are used only when errors are detected.

FIGURE 4-12

Fraction of lives with no arithmetic
Class Noari

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.213	0.637	0.574	0.494	0.470
Crout	0.528	0.716	0.214	0.349	0.440
Treesort	0.315	0.686	0.257	0.784	0.565
PERT	0.597	0.735	0.547	0.457	0.416
Håvie	0.628	0.680	0.482	0.496	0.412
Ising	0.695	-	0.620	0.744	0.622
Secant	-	-	-	0.263	0.266

Algorithm\Programmer	E	B	A	G	L
Aitken	0.317	0.390	0.402	0.475	0.391

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.844	0.744	0.921	0.802	0.886

FIGURE 4-13

Fraction of lives with fixed point add/subtract
Class Counter

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.504	0.106	0.054	0.118	0.141
Crout	0.304	0.009	0.096	0.189	0.122
Treesort	0.355	0.103	0.710	0.208	0.056
PERT	0.380	0.122	0.397	0.516	0.552
Håvie	0.278	0.085	0.149	0.123	0.156
Ising	0.300	-	0.373	0.250	0.370
Secant	-	-	-	0.359	0.303

Algorithm\Programmer	E	B	A	G	L
Aitken	0.210	0.202	0.302	0.423	0.389

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.130	0.234	0.074	0.190	0.108

FIGURE 4-14

Fraction of lives with fixed point multiply/divide
Class Fixed

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.009	0.001	0.018	0.042	0.019
Crout	0.006	0.064	0.433	0.156	0.142
Treesort	0.317	0	0.011	0.000	0.370
PERT	0.002	0.000	0.004	0.006	0.006
Håvie	0.002	0.001	0.031	0.018	0.015
Ising	0.006	-	0.007	0.006	0.008
Secant	-	-	-	0.175	0.199

Algorithm\Programmer	E	B	A	G	L
Aitken	0	0	0	0	0.085

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.026	0.019	0.005	0.009	0.008

FIGURE 4-15

Fraction of lives with floating point arithmetic
Class Floating

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.274	0.256	0.354	0.347	0.369
Crout	0.163	0.211	0.257	0.306	0.296
Treesort	0.014	0.211	0.022	0.008	0.009
PERT	0.021	0.143	0.053	0.021	0.026
Håvie	0.092	0.233	0.339	0.363	0.418
Ising	0.000	-	0	0	0
Secant	-	-	-	0.203	0.232

Algorithm\Programmer	E	B	A	G	L
Aitken	0.473	0.408	0.296	0.102	0.136

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.000	0.003	0	0	0

FIGURE 4-16

Fraction of R-lives used as temporaries only
Class Temporary

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.028	0.067	0.179	0.101	0.121
Crout	0.018	0.101	0.049	0.137	0.142
Treesort	0.001	0.107	0.000	0.000	0.001
PERT	0.016	0.128	0.188	0.069	0.104
Håvie	0.072	0.279	0.062	0.250	0.019
Ising	0.059	-	0.086	0.147	0.067
Secant	-	-	-	0.041	0.030

Algorithm\Programmer	E	B	A	G	L
Aitken	0.062	0.078	0.092	0.112	0.015

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.096	0.089	0.180	0.151	0.153

FIGURE 4-17

Fraction of lives used for indexing
Class Indexing

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.513	0.407	0.226	0.341	0.251
Crout	0.519	0.374	0.520	0.195	0.244
Treesort	0.482	0.412	0.683	0.431	0.476
PERT	0.592	0.421	0.556	0.445	0.497
Håvie	0.524	0.365	0.387	0.278	0.203
Ising	0.571	-	0.484	0.267	0.249
Secant	-	-	-	0.376	0.406

Algorithm\Programmer	E	B	A	G	L
Aitken	0.185	0.196	0.232	0.318	0.474

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.401	0.364	0.341	0.509	0.313

2) A significant fraction of the R-lives are never stored. This hypothesis was verified for all subject programs. It clearly demonstrates that registers are not only needed to produce results, but also as indices and fast temporary storage.

3) The ***usage classes*** representing most lives have few ***attributes***, i.e. 2 or 3. This hypothesis was verified in all subject programs. It supports the idea put forward by Knuth [KnuD70], that programmers rarely do anything complicated.

4) Most lives for indexing use no arithmetic at all. This was true in most cases, but with notable exceptions.

5) Most lives used for indexing have no arithmetic stronger than fixed point add and subtract. Largely verified, but strong exceptions. Particularly noteworthy was the Crout algorithm, the only one where two dimensional arrays were used. There was a great difference between programs using a multiplicative address calculation (dope vectors) (FORTRAN and BLISS versions) and those using Iliffe vectors (ALGOL version) for array accessing.

6) Lives used for floating point arithmetic rarely use fixed point arithmetic. True for all subject programs that have a significant amount of floating point arithmetic. The indications were that the exceptions were usages for fixed to floating conversion or vice versa, largely occuring in the initialization phases of our programs.

Another observation was that most ***usage classes***, although not the most frequent ones, contained the "tested" ***attribute***.

An obvious source of error with this method is its dependence on the correct detection of R-lives, as discussed on page 49. As noted there, this error may be significant for some of our ALGOL programs.

Another deficiency is that the representation of a ***usage class*** does not take into account that some ***attributes*** may contribute to the class many more times than others. The algorithm could be augmented to compute the number of occurrences of each ***usage attribute*** while accumulating the class of an R-life. Even if these counts were averaged over the lives in each ***usage class***, one word of storage would be required for each combination of attribute and ***usage class***, i.e. at least 4000 words. Since most lives are short and of few usages, we believe that this addition to the algorithm does not justify its cost. We believe that the trend of such results would be that the infrequent events are even less frequent than shown by our present methods.

4.5.1 Summary

The results in figures 4-16 to 4-15 lead us to the following conclusions:

For algorithms containing floating point arithmetic, up to 42% of the R-lives are from the "Floating" class, but usually considerably fewer: 20% to 37%. The BASIC programs form an exception, even though all arithmetic in BASIC is done in floating point, at most 26% of the R-lives are from this class. Except for BASIC programs, there is a systematic variation with the algorithm.

Lives with fixed point multiplication and division occur almost only in the programs that use the multiplicative method for matrix access, or that use integer division for unpacking. Hence the dependence on algorithm is marked, but less so than for the "Floating" class, and particular techniques used by or enforced by the language or its implementation become significant.

For the other classes, the interaction of the needs of the algorithm with the register allocation mechanism of the compilers obscure any systematic effects due to each of these factors singly. There is, however, some more stability to the results from the ALGOL and BASIC programs than from the others. This is most probably due to the run time system of ALGOL and to the lack of integer arithmetic in BASIC.

ALGOL programs have a high number of lives in the "Counter" class, (30% to 50% of the lives); BASIC programs have a very large number of lives with no arithmetic (63% to 74%). ALGOL programs also have a high number of lives in this class (21% to 69%).

48% to 59% of the R-lives in ALGOL programs are used for indexing. The fraction of indexing lives is also high in BLISS programs (23% to 68%) and BASIC programs (37% to 42%), but not consistently. For the FORTRAN programs this fraction varies between 19% and 49%, the agreement between the two FORTRAN versions is good.

For The "Temporary" class, the results vary between 0 and 28%. For ALGOL programs the results are consistently low, 0.1% to 7.2%. For BASIC programs they are high: 6.7% to 28%.

The substance of these results is: The classes for strong arithmetic are used only if the algorithm or the accessing method used by the compiler requires such arithmetic. Hence for these classes the dependence on the algorithm is strong. In the classes for weak and no arithmetic the results seem to depend more on the language, particularly for those languages which enforce a strong regimen on their programs, such as ALGOL by its run time system and BASIC by its restriction to floating arithmetic and by its strictly statement by statement execution (no information is carried in registers between source program lines).

These findings corroborate those of Alexander [AleW72], which indicate that two or three of the physical registers on the IBM 360 are used as accumulators, whereas most of them are used as indices or base registers.

The results for the FORTRAN and BLISS programs show little systematic variation except for a good agreement between the FORTRAN versions of the same algorithm.

4.6 Register block size

The results presented in Figure 4-9 through Figure 4-11 indicate that for our subject set the number of register references is between two and three times the number of memory references. Hence the need for a register block is well demonstrated by experiment, as well as being motivated by programmer experience. The problem is more one of size, i.e. how many registers can be utilized efficiently enough to warrant their cost. In addition to its obvious dependence on the other properties of the ISP, this number depends on the structure of the algorithm, the cleverness of the programmer and the compiler and the fineness of the factorization of the program. The combined effect of these factors is represented by our subject set.

We now present a sequence of methods which in a gradually better way measure the utility of the register block and the time costs associated with its usage.

We have already presented some crude measures in Section 4.4: The number of memory and register references per instruction presented in figures 4-9 through 4-11 are of relevance, another measure is the average number of *live* registers in Figure 4-8.

Some better measures could be developed if we knew the number of registers that are are *live* at each point in the program. In the next subsection we present an algorithm for computing this. This algorithm is extended to compute, for any N, what fraction of the time at least N registers were *live*, and finally to give a coarse estimate of the time cost incurred if the number of registers were reduced below the maximum used by the program. This estimate is based on the number of usages in each R-life. A further improvement takes into account long *dormant* periods of registers. We now describe these algorithms, the associated cost measures, and the experimental results, in more detail.

4.6.1 Detecting simultaneous lives

The algorithms are embodied in a two stage (or pass) program, the first stage reads the trace and writes an intermediate file of data items describing each R-life. This file is processed in the reverse order by the second stage. The algorithms are described below, and illustrated by an example in Figure 4-1?.

The first stage is actually the algorithm which detects register lives, described in Section 4.4, with a minor addition: As each R-life is determined, (at the start of the next R-life for that register), a data item containing the times of its *transitions*, its *usage class*, number of usages etc. is written to the intermediate file.

The second stage reads this file backwards while maintaining a simulated time (s-time) which decreases as the algorithm proceeds. Initially the s-time is the duration of the program, later it is equal to the time of the *transition* most recently processed by the algorithm as described below.

The stage two program keeps a data entry describing the state (*live* or *dead*) of each physical register, there is also a counter of *live* registers, and a linked list of at most two entries (each describing an unprocessed *transition*) per physical register, as described below.

Initially the second stage reads the data items decribing the last R-life for each register, and enters the *transitions* in the list, sorted by decreasing time. The algorithm proceeds by processing the *transition* first on the list, i.e. that having the highest time. Current s-time is set to this time, and the table and counter are updated according to the nature of the *transition*. If the *transition* was a *first use*, we have finished processing an R-life. The next data item for that register is immediately read from the file (see below), and its *transitions* are entered in the list. Hence when the analysis is under way, the list contains one *transition* for each *live* register, (i.e. its *first use*), and both *transitions* for the other registers (whose data items have been read, but whose times of *last use* are less than the current s-time).

Note that, by the way the intermediate file was written, its data items are ordered by the time of *first use* of the next (later in execution time) R-life of the register involved. When the file is read backwards by stage 2, one item is read each time a *first use* has been processed. The item read is the one that was output by stage one at that point of the trace when the execution time of the subject program was equal to the current s-time. But that is exactly the data item describing the next (earlier in execution, lower s-time) R-life for the register just processed by stage 2. An exception may occur when the same instruction *loaded* two registers, and hence started two R-lives, in which case their order in the file may be the reverse of what stage 2 expects. Consequently data space is needed to describe in full exactly one R-life for each physical register, plus one extra R-life possibly being held over for one read operation. This is further illustrated in Figure 4-18. The order of events during the interval described by the figure is:

<u>During execution</u>:

Before T0: R0, R2 and R3 are live.

At T0: R1 is loaded, L10 starts. R3 is accessed.

At T1: R0 is loaded using R0 as index. Hence L00 and L01 overlap at T1.

At T2: Last usage of L01 and L20.

At T3: Last usage of L10; R0 is loaded; hence L02 starts. R3 is accessed for the first time since T0.

At T4: Last usage of L30; R1 is loaded; hence L11 starts.

At T5: Both R2 and R3 are loaded by the same instruction. L21 and L31 start.

At T6: Last use of L11.

After T6: R0, R2 and R3 are live.

<u>During stage 1</u>:

At T1: L00 is detected and its data item output.

At T3: L01 is detected and its data item output.

At T4: L10 is detected and its data item output.

At T5: L20 and L30 are detected and their data items output in some order.

We denote the data items DLij etc. The data items on the intermediate file are now in the order:

. . . DL00 DL01 DL10 DL20 DL30 . . .

The two last might be interchanged; we assume this order.

<u>During stage 2</u>: (Listed in order of occurrence in stage 2, i.e. by decreasing s-time).

S-time > T6: The data items DL02, DL21 and DL31 have been read and the last usages of their lives processed. DL11 has been read but its transitions have not yet been processed.

S-time = T6: Last use of L11 is processed.

S-time = T5: First usages of L21 and L31 are processed, assume in that order. After L21 has been processed a data item is read. By the above assumptions this is DL30. Hence it will be held over in temporary storage, and DL20 is read from the file, and entered into the tables. Next the first usage of L31 is processed and DL30 is fetched from the temporary store and entered in the tables.

S-time = T4: The first use of L11 is processed and DL10 is read from the file. The last use of L30 is processed.

S-time = T3: The first use of L02 is processed, and the data item DL01 is read. The last use of L10 is processed.

S-time = T2: The last uses of L01 and L20 are processed.

S-time = T1: The first use of L01 is processed, the data item DL00 is read and its last use immediately processed.

S-time = T0: The first use of L10 is processed, the data item for its previous life, if any, is read.

Now assume R3 was dormant from T0 to T3. This would be detected by stage 1 at time T3, the data item for the first part of L30 (call it DL30') would be output at this time. The data item for the second part of L30 (i.e. DL30") would be output at T5, as was DL30. During stage 2, the data item DL30" would be read at s-time T5, its usages processed at T4 and T3. At T3 the data DL30' would be read, its last usage would be processed at T0, and so on as before.

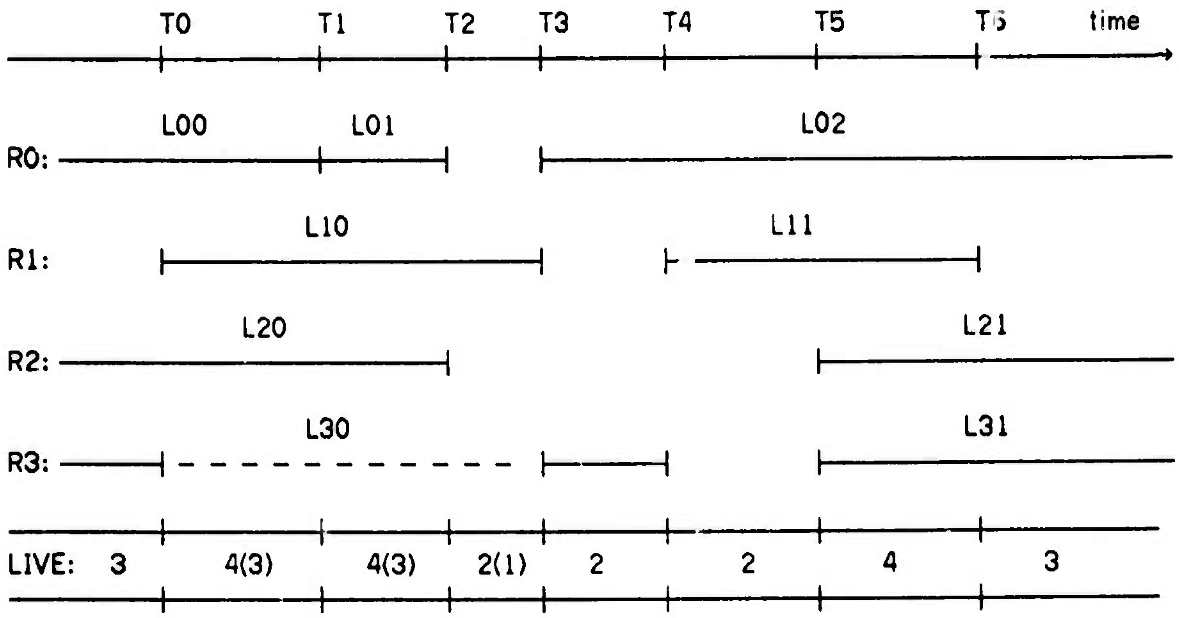
For each interval of time, the number of live registers is given at the bottom of the diagram. In the latter case it would be reduced by 1 between T0 and T3.

This concludes our discussion of Figure 4-18.

FIGURE 4-18

A typical situation of Register usage.

Assume our ISP has four registers, R0, R1, R2, R3. The successive lives of Ri are denoted Li0, Li1, ... The diagram has one horisontal line for each register, as labelled. This line is solid when that register is live. It is broken when that register is dormant. The vertical bars correspond to times of transistion, as marked on the time axis at the top.



The *usage class* of each R-life may be included in each data item on the intermediate file. Hence, if the result of an analysis as described in Section 4.5 should indicate that specialization of the registers is desirable we may do this simultaneity determination for any *usage class* we consider important, in addition to the set of all registers. The "state" of each physical register has to be augmented to include its class, and an encoding of this class into the (probably much fewer) classes for which output is desired must be deviced. For each output class a counter of *live* registers must be added.

We performed these analyses for the subclasses of R-lives defined in Section 4.5, as well as for the class of all registers. A typical output from phase 2 is displayed in Figure 4-19. A compressed form of the results from all the subject programs is given in figures 4-20 through 4-22.

FIGURE 4-19

Output from simultaneously live register analysis for program FORTEN Håvie.
Distribution of number of live registers in the different classes.

For each class, the first coloumn gives the instruction count when exactly N registers were live. Coloumn 2 gives the fraction of the total instruction count for this state. Coloumn 3 is a cumulation of coloumn 2, it gives the fraction of the instruction count when at most N registers were live.

N	NO ARITHMETIC			FIXPOINT ADD/SUB.			FIXPOINT MUL/DIV.			N
1	25221	0.693	0.693	1960	0.054	0.054	410	0.011	0.011	1
2	7680	0.211	0.904	23837	0.655	0.709	215	0.006	0.017	2
3	1163	0.032	0.936	7460	0.205	0.913	14	0.000	0.018	3
4	1038	0.029	0.964	198	0.005	0.919	0	0.000	0.018	4
5	551	0.015	0.979	254	0.007	0.926	0	0.000	0.018	5
6	433	0.012	0.991	134	0.004	0.930	0	0.000	0.018	6
7	256	0.007	0.998	5	0.000	0.930	0	0.000	0.018	7
8	41	0.001	0.999	0	0.000	0.930	0	0.000	0.018	8
9	47	0.001	1.001	0	0.000	0.930	0	0.000	0.018	9
10	14	0.000	1.001	0	0.000	0.930	0	0.000	0.018	10
11	0	0.000	1.001	0	0.000	0.930	0	0.000	0.018	11
12	0	0.000	1.001	0	0.000	0.930	0	0.000	0.018	12
13	0	0.000	1.001	0	0.000	0.930	0	0.000	0.018	13
TOTALS										
13	36444		1.001	33848		0.930	639		0.018	13

N	FLOATING POINT			INDEXING			ANY USAGE			N
1	18172	0.499	0.499	28218	0.775	0.775	166	0.005	0.005	1
2	6446	0.177	0.676	5853	0.161	0.936	1104	0.030	0.035	2
3	34	0.001	0.677	350	0.010	0.945	3171	0.087	0.122	3
4	0	0.000	0.677	426	0.012	0.957	14985	0.412	0.534	4
5	0	0.000	0.677	718	0.020	0.977	15092	0.415	0.948	5
6	0	0.000	0.677	515	0.014	0.991	481	0.013	0.961	6
7	0	0.000	0.677	335	0.009	1.000	298	0.008	0.969	7
8	0	0.000	0.677	45	0.001	1.001	409	0.011	0.981	8
9	0	0.000	0.677	18	0.000	1.002	419	0.012	0.992	9
10	0	0.000	0.677	0	0.000	1.002	185	0.005	0.997	10
11	0	0.000	0.677	0	0.000	1.002	78	0.002	0.999	11
12	0	0.000	0.677	0	0.000	1.002	50	0.001	1.001	12
13	0	0.000	0.677	0	0.000	1.002	47	0.001	1.002	13
TOTALS										
13	24652		0.677	36478		1.002	36485		1.002	13

FIGURE 4-20

Maximal number of simultaneous R-lives
Number of registers sufficient 98% of the time
Number of registers sufficient 90% of the time

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	max	13	10	9	13	12
	98%	11	7	6	10	9
	90%	8	6	5	9	7
Crout	max	13	7	7	13	12
	98%	11	7	7	12	8
	90%	10	6	6	10	7
Treesort	max	14	7	6	4	12
	98%	4	7	5	4	5
	90%	3	6	5	3	4
PERT	max	14	10	7	11	12
	98%	10	7	6	8	8
	90%	8	6	5	3	5
Håvie	max	14	10	9	10	13
	98%	11	6	6	6	9
	90%	9	5	5	5	5
Ising	max	14	-	7	11	12
	98%	11	-	5	7	9
	90%	10	-	5	3	6
Secant	max	-	-	-	13	12
	98%	-	-	-	6	6
	90%	-	-	-	5	5

Algorithm\Programmer		E	B	A	G	L
Aitken	max	7	7	8	7	8
	98%	7	7	7	7	7
	90%	7	6	6	6	7

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	max	15	11	13	13	11
	98%	10	9	6	8	8
	90%	8	7	5	7	6

FIGURE 4-21

Number of registers sufficient 90% of the time
for the arithmetic classes previously defined. Classes denoted by
FLO = Floating, FIX = Full fixpoint, COU = Fixpoint add subtract.

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	FLO	2	1	2	2	2
	FIX	1	0	0	1	0
	COU	4	2	2	1	2
Crout	FLO	1	1	1	3	2
	FIX	0	1	2	4	2
	COU	5	1	3	3	3
Treesort	FLO	0	1	0	0	0
	FIX	1	0	0	0	1
	COU	1	2	3	1	2
PERT	FLO	0	1	1	0	0
	FIX	0	0	0	0	0
	COU	4	2	3	2	3
Håvie	FLO	1	2	2	2	2
	FIX	0	0	1	0	0
	COU	5	2	2	2	3
Ising	FLO	0	-	0	0	0
	FIX	0	-	0	0	0
	COU	5	-	4	1	3
Secant	FLO	-	-	-	2	1
	FIX	-	-	-	1	1
	COU	-	-	-	2	4

Algorithm\Programmer		E	B	A	G	L
Aitken	FLO	2	2	2	2	2
	FIX	0	0	0	0	1
	COU	3	2	3	4	3

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	FLO	0	0	0	0	0
	FIX	0	1	0	0	0
	COU	3	2	2	2	2

FIGURE 4-22

Number of registers sufficient 90% of the time
for the no arithmetic class (NOA), the indexing class (IND)
and the total class (TOT).

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	NOA	4	4	3	7	5
	IND	6	3	2	5	5
	TOT	8	6	5	9	7
Crout	NOA	6	4	2	3	5
	IND	9	3	3	2	3
	TOT	10	6	6	10	7
Treesort	NOA	2	4	2	2	2
	IND	2	3	3	2	2
	TOT	3	6	5	3	4
PERT	NOA	4	4	2	2	3
	IND	7	3	3	2	2
	TOT	8	6	5	3	5
Håvie	NOA	5	3	2	2	2
	IND	8	3	2	2	2
	TOT	9	5	5	5	5
Ising	NOA	6	-	2	2	4
	IND	9	-	2	2	4
	TOT	10	-	5	3	6
Secant	NOA	-	-	-	2	2
	IND	-	-	-	2	2
	TOT	-	-	-	5	5

Algorithm\Programmer		E	B	A	G	L
Aitken	NOA	4	4	4	3	2
	IND	4	3	3	2	5
	TOT	7	6	6	6	7

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	NOA	6	5	4	6	4
	IND	4	4	2	4	2
	TOT	8	7	5	7	6

4.6.2 Cost of reducing the register block

The results just presented show clearly that, except for ALGOL programs and the ALGOL compiler, at most 8 to 10 registers out of the 16 available are used simultaneously†, and that many only for short intervals of time. If the processor were equipped with fewer registers than this, a time and space cost would occur by having to store registers temporarily in primary memory. Intuitively, it seems from the above results that for a moderate reduction in the number of registers this cost would be low. We now describe an extension to our algorithm which enables us to compute upper bounds for this time cost.

Assume we want to compute the additional time cost incurred by running the program on an ISP with M registers but otherwise similar to the one we investigate. At some point in the program we have N simultaneous lives, N > M. We select the N - M least useful lives as described below, and assume that these can be <u>interleaved</u> with the remaining R-lives in the registers used for the latter lives. That is: Each time an omitted register is referenced, another register must be temporarily stored, and the desired value loaded into it. This value is stored after use, and the original value reloaded. The associated time cost is two STORE LOAD pairs per reference to the selected lives, i. e. 4 instructions per reference if the instruction count is used. If an R-life L so selected for omission, is selected again at some later time, but for the same M, the cost should not be added the second and later times.

This computation is done during the second stage described above, each time we process a ***first use***. It can be done simultaneously for all desired M, and for many criteria of usefulness of lives. Data space used by the algorithm is proportional to the number of criteria times the number of registers, but with a low factor (at most 5 words). The amount of computation

- - - - - - -

† The structure of an ALGOL program is almost like two coroutines calling each other, viz. the user program and the run time support routines. These operate on disjoint memory cells and almost disjoint sets of registers. Similarly the ALGOL compiler consists of a lexical analyser, a syntax analyser and a code generator, each having its own set of registers allocated to it. This probably accounts for the exceptional results obtained for ALGOL, and also indicates how programs may be structured to use many registers effectively. Further explanation may be the difficulty of detecting multi-instruction ***loads***, as described on page 49.

involved is small. Hence this is a relatively cheap measure to compute once we are doing the simultaneity analysis.

Several criteria of usefulness can be used to select which R-lives to omit. The following were tried:

The least used lives.

The least densely used lives (usages per lifelength).

The shortest lives.

The longest lives. (Might be better than omitting many short ones).

Of these, the "longest lives" never gave the lowest cost. The "shortest lives" criterion rarely gave good results. Almost all the lowest results were obtained using the "least used" or "least densely used" criteria. Furthermore the criterion giving the lowest cost often changed with the number of available registers (i.e. M) even for the same program. It follows that, in an analysis, several criteria should be used, including the 3 first ones above. The best cost obtained in each case should then be used as an upper bound.

We present a typical output in Figure 4-23, and a summary of the results from the whole subject set in Figure 4-24. As is seen, the cost of reducing the number of registers in most cases is low, less than a percent in some cases, and less than 15% in most, but running very high in a few cases (70% - 100% increase in cost). We investigate this further below.

Note that 3 of the programs which give extremely high costs are ALGOL programs, and just those which have many procedure calls and parameter transmissions. Hence the arguments presented above about the coroutine like structure of ALGOL programs, and also the error discussed on page 49 in connection with undetected *loads*, apply with force to these results.

4.6.3 Some sources of error

We now discuss some sources of errors associated with this method.

The most significant is probably that the lives omitted are selected on basis of their average properties. A better selection might have been made, had the local properties of lives been known. We discuss below how this can be done.

FIGURE 4-23

Cost of reducing number of available registers.
Lives with lowest utility are omitted, 4 utility criteria are used.
Sample output from program FORTEN Håvie.

UTILITY: REFERENCES IN LIFE

# OF REGS	OMITTED ACCESSES	RELATIVE MAX COST	LIVES OMITTED
12	33	0.0036	17
11	98	0.0108	42
10	155	0.0170	66
9	227	0.0249	92
8	409	0.0449	167
7	659	0.0724	256
6	1077	0.1183	361

UTILITY: DENSITY OF REFERENCES

# OF REGS	OMITTED ACCESSES	RELATIVE MAX COST	LIVES OMITTED
12	7	0.0008	2
11	27	0.0030	6
10	58	0.0064	12
9	2386	0.2621	19
8	2500	0.2747	29
7	2704	0.2971	39
6	2883	0.3167	51

UTILITY: LENGTH OF LIFE

# OF REGS	OMITTED ACCESSES	RELATIVE MAX COST	LIVES OMITTED
12	33	0.0036	17
11	122	0.0134	45
10	206	0.0226	70
9	356	0.0391	108
8	700	0.0769	202
7	1014	0.1114	294
6	1342	0.1474	382

UTILITY: SHORTNESS OF LIFE

# OF REGS	OMITTED ACCESSES	RELATIVE MAX COST	LIVES OMITTED
12	2410	0.2648	2
11	2591	0.2847	4
10	2713	0.2981	8
9	2815	0.3093	12
8	2888	0.3173	16
7	3009	0.3306	24
6	3170	0.3483	36

FIGURE 4-24

Upper bound for time cost of reducing the register block
to 10, 8 or 7 registers respectively,
given as relative increase in instruction count.

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	10 rg	0.054	0	0	0.013	0.005
	8 rg	0.228	0.001	0.000	0.132	0.091
	7 rg	0.368	0.002	0.004	0.250	0.180
Crout	10 rg	0.076	0	0	0.440	0.000
	8 rg	0.384	0	0	0.757	0.006
	7 rg	0.772	0	0	1.046	0.081
Treesort	10 rg	0.002	0	0	0	0.000
	8 rg	0.005	0	0	0	0.001
	7 rg	0.007	0	0	0	0.001
PERT	10 rg	0.016	0	0	0.003	0.003
	8 rg	0.132	0.000	0	0.035	0.037
	7 rg	0.212	0.001	0	0.052	0.066
Håvie	10 rg	0.060	0	0	0	0.006
	8 rg	0.575	0.001	0.001	0.004	0.045
	7 rg	0.734	0.003	0.006	0.017	0.072
Ising	10 rg	0.067	-	0	0.000	0.004
	8 rg	0.437	-	0	0.008	0.051
	7 rg	0.997	-	0	0.029	0.105
Secant	10 rg	-	-	-	0.001	0.002
	8 rg	-	-	-	0.009	0.014
	7 rg	-	-	-	0.015	0.020

Algorithm\Programmer		E	B	A	G	L
Aitken	10 rg	0	0	0	0	0
	8 rg	0	0	0	0	0
	7 rg	0	0	0.011	0	0.003

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	10 rg	0.018	0.001	0.000	0.003	0.001
	8 rg	0.068	0.037	0.002	0.062	0.009
	7 rg	0.121	0.082	0.010	0.215	0.023

Furthermore a program written for an ISP with few registers will be quite different in its local structure from a program written with a large register block in mind. Hence this method can not be used to estimate the cost of large reductions in register block size. One would also, a priori, believe this argument to hold for reduction to a relatively small number of registers even if the program did not use many in the first place. This belief, however, is not vindicated by our results.

For the same reason we would expect the upper bounds found by this algorithm, and by its modified version described below, to be considerably higher than the actual cost obtained by average to careful recoding for the lower number of registers.

A third source of errors is that successive lives of the same register may overlap by one instruction†, hence the simulation of two lives in one register may not be valid. We have counted the number of such overlaps and found it mostly to be small (see Figure 4-25). Hence this source of errors is insignificant.

Finally our simulation might be invalid because there were not enough registers available to hold the necessary lives. Since at most 4 registers can be involved by any PDP-10 instruction, this error will not occur for $M > 4$. We never used $M < 6$.

4.6.4 Utilizing dormant periods

We now consider a way to take local behaviour of registers into account when computing the cost of running with a smaller register block. This is done by assuming that a register is ***dead*** whenever it has been ***dormant*** for some time K. If this assumption should be wrong, a time cost of one STORE, LOAD pair applies for each R-life prematurely terminated based on the assumption.

We can detect such dormant periods during the first stage of the analysis. Each time a

- - - - - - -

† As when ***loading*** a register using the same register in the address calculation (MOVE RG,FLOP(RG)). If we had used a finer grain of time, as discussed in Section 4.2, this problem could have been avoided.

register is used, it is easily checked if its previous usage was more than K ago. If so, the present usage is processed as a *load*, and a "prematurely killed" counter is updated.

The effect of this trick is that a register will appear to be *dead* whenever it has a long dormant period. Hence during this apparently *dead* period, the number of live registers is reduced by one. Non overlapping R-lives of other registers, occurring within this period, can be accomodated in the apparently dead register at no cost beyond that of saving and restoring the *dormant* life once (i.e. one STORE LOAD pair). This cost is at most half of the cost of interleaving any two lives, and independent of how many other lives are accomodated in the dormant register. Since most R-lives are short, we would expect a considerable decrease of cost to be obtained this way. However, since each choice of K requires a separate intermediate file, at least logically, and the simultaneity determination has to be done for each of these, it is a more costly analysis to apply.

An alternative approach is to use a hybrid method, - some reasonable K is chosen for phase one, and the interleaving process is applied in phase 2. If the cost so obtained seems unreasonably high, a new analysis can be run using a smaller K.

For our experiments we used this hybrid method. Unless otherwise specified, K was chosen to be 200 throughout all the experiments. The number of lives prematurely terminated by this assumption is tabulated in Figure 4-26. Note that if the same life has several dormant periods of length more than K, each non dormant period is counted as a life.

To see the effect of varying K, we performed some experiments with K=100, K=60, K=40 and K=25. For this purpose we chose programs that gave particularly high cost with K=200, in the hope that cost could be reduced this way. The programs chosen were the ALGOL versions of Ising, Håvie and Crout, and the FORFOR version of Crout. For comparison we also included two programs where the analysis algorithm performed well, i. e. where the results for K=200 were regular and the costs low. These were the FORTEN versions of Håvie and Crout. The results are displayed in Figure 4-27.

The overall trend of these results is that the upper bound of the cost can be reduced considerably by using a small K. However, there is a point where the cost from storing and restoring *dormant* lives becomes comparable to the cost of *interleaving* lives, and the total cost rises. This point is higher (larger K) the lower the cost of interleaving. We have at present no mechanical way of guessing what K will be optimal for a given program without performing a series of experiments. By choosing K as low as 25, the cost of reducing the

FIGURE 4-25

Fraction of lives overlapping their successor

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.275	0.101	0.005	0.066	0.071
Crout	0.190	0.135	0.028	0.113	0.136
Treesort	0.155	0.103	0.050	0.002	0.097
PERT	0.199	0.030	0	0.066	0.341
Håvie	0.110	0.020	0.000	0.132	0.010
Ising	0.106	-	0.022	0.074	0.013
Secant	-	-	-	0.035	0.042

Algorithm\Programmer	E	B	A	G	L
Aitken	0	0	0.004	0.001	0.002

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.038	0.020	0.002	0.044	0.003

Computed as: (number of overlaps)/(number of lives).

FIGURE 4-26

Lives prematurely terminated by 200 instructions dormancy rule

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	45	8	64	39	35
Crout	37	15	126	16	156
Treesort	14	1	579	2	461
PERT	35	3	5	11	8
Håvie	15	11	21	17	8
Ising	54	-	66	24	13
Secant	-	-	-	805	795

Algorithm\Programmer	E	B	A	G	L
Aitken	63	63	72	99	135

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	489	141	799	2819	1035

register block was dramatically reduced for those programs where this cost previously was high. The increase in instruction count for reducing to 7 registers was in all cases but one brought below 20%. We believe the cost for this program could be brought further down by using even lower K.

The cost obtained by any of these methods is an upper bound, hence we may safely assume the smallest of them to be a valid upper bound.

4.6.5 Summary

The maximal number of registers used simultaneously by any of our 41 subject programs is 15. For 17 programs it is 10 or less. 10 registers would suffice 90% of the time (instruction count) for all the programs, 98% of the time for 36 of them. 8 registers would suffice 90% of the time for 36 programs, 98% of the time for 29 programs.

BLISS programs use the fewest registers, BASIC programs also use few. Hence time efficient programs do not necessarily use many registers. ALGOL programs use most registers, but not more than maximally used by FORTRAN programs. The compilers use no more registers than the small programs, and the reduction costs for the compilers are not significantly higher than for the small programs. Hence the size and complexity of the program has little influence on these results.

The results for the individual classes show that 90% of the time 2 floating point accumulators would be sufficient for all the programs, 1 register with full fixpoint abilities would be sufficient except for the FORFOR version of Crout, and 5 registers with fixpoint addition and subtraction would suffice for all programs. Similarly, 7 registers without arithmetic capabilities and 9 indexing registers would be sufficient 90% of the time for all the programs.

All the above results are obtained on the assumption that a register is *dead* when it has been *dormant* for 200 instructions. Our experiments using a reduced such period indicate that lower results would be obtained that way.

If the register block were to be reduced to 8 registers, the increase in instruction count would be less than 5% in 30 of the programs, less than 20% in 36 of them. Again the results

FIGURE 4-27

Relative increase o' instruction count by interleaving R-lives as a function of K and M, for selected subject programs.

Algorithm	Maximal dormancy	ALGOL Ising	ALGOL Håvie	ALGOL Crout	FORFOR Crout	FORTEN Crout	FORTEN Håvie
Lives added by dormancy saving	200	54	15	37	16	156	8
	100	417	65	320	224	324	29
	60	614	129	334	255	602	65
	40	1055	1218	509	3692	611	108
	25	5158	7663	5007	4931	2561	2299
Dormancy part of relative increase	200	0.001	0.000	0.001	0.000	0.005	0.000
	100	0.009	0.001	0.006	0.004	0.010	0.002
	60	0.013	0.002	0.006	0.005	0.019	0.004
	40	0.023	0.014	0.009	0.067	0.019	0.006
	25	0.113	0.091	0.087	0.090	0.081	0.126
Total increase for reduction to 10 registers	200	0.068	0.060	0.077	0.440	0.005	0.006
	100	0.048	0.054	0.009	0.402	0.001	0.004
	60	0.041	0.054	0.008	0.403	0.019	0.004
	40	0.023	0.015	0.009	0.082	0.019	0.006
	25	0.113	0.091	0.087	0.090	0.081	0.126
Total increase for reduction to 8 registers	200	0.438	0.575	0.385	0.757	0.011	0.045
	100	0.410	0.558	0.270	0.731	0.012	0.026
	60	0.349	0.556	0.269	0.732	0.019	0.011
	40	0.316	0.269	0.254	0.277	0.019	0.010
	25	0.121	0.094	0.088	0.179	0.081	0.126
Total increase for reduction to 7 registers	200	0.998	0.734	0.773	1.046	0.086	0.072
	100	0.627	0.714	0.411	0.999	0.046	0.042
	60	0.577	0.710	0.410	1.000	0.041	0.030
	40	0.522	0.674	0.377	0.494	0.041	0.016
	25	0.190	0.149	0.144	0.269	0.082	0.127

are based on maximal dormant periods of 200 instructions. Additional experiments, using 4 of the programs where reduction was most costly, show that by reducing this period to 25 instructions the costs were reduced from 44%, 58%, 38% and 76% to 12%, 9.4%, 9% and 18% respectively, for these 4 programs. We did not investigate if a further reduction to 20 or 15 would reduce the cost further.

The cost is particularly high for ALGOL programs. This is discussed in a footnote on page 74. FORFOR Crout also has a high cost, and its cost was the hardest to reduce by decreasing the maximal dormancy. For BLISS and BASIC programs the reduction was particularly cheap, less than 1% for each program, including the two compilers written in BLISS. The correlation between the two FORTRAN versions is not particularly good.

4.7 Utilities of values

The methods just described are aimed at establishing the effect of reducing the register block, and our experiments indicate that the registers on the whole are not used very efficiently. However, there might be values in memory that could benefit by being kept in registers if the programmer or compiler had realized it. Hence it would be desirable to have a utility measure which indicates what values are most important, locally in time, at each point in the computation. Those values should be kept in registers which have the highest utility at that point in time. Further if values of high utility can not be held in registers, we have an indication that more registers should be included in the processor. The converse holds if only a few values have high utility.

Such a measure must give greatest importance to values used by the current instruction, less weight to values used further away in the instruction stream. The function w(s) below is intended to express this. Furthermore to simplify computations, we might not want to consider all accesses to a value, only those within some interval of time containing the current instruction execution. This is expressed by the function i(s).

A class of such measures can be defined as follows: Define the <u>utility</u> of a value V at time t to be:

$$P(V,t) = \int_0^\infty w(T-t) * i(T-t) * u(V,T)\, dT$$

where

$w(s)$ is a weighting function

$i(s)$ is 1 in the interval considered, 0 elsewhere

$u(V,t)$ is 1 if V was used by an instruction executed at time t,
0 otherwise.

$w(s)$ and $i(s)$ can be chosen freely to obtain different measures of utility, whereas $u(V,t)$ is a formalization of the trace. In choosing $w(s)$ and $i(s)$ one must take care that values used by the current instruction get a higher utility than any other, regardless of how much they are used in the surrounding interval.

It is reasonable to use the instruction count as the time measure rather than the computed time. Some tentative choices for interval functions can then be classified as:

[n,m] : $i(s)$ = 1 for the interval containing the last n and next m uses of the value,
0 otherwise.

(n,m) : $i(s)$ = 1 for the last n and next m instructions,
0 otherwise.

One such measure could be defined as follows:
Let k be the next time value V will be used, i.e.:

$u(V,T) = 0$ for T in [t,k>,

$u(V,k) = 1$ for $T = k$,

$u(V,k)$ is irrelevant otherwise.

Now let

$i(s) = 0$ for $s < 0$ $(T < t)$

$i(s) = 1$ for $k \geq s \geq 0$

$i(s) = 0$ for $s > k$

and let

$w(s) = 1/(|s| + 1)$

I.e. $P(V,t)$ is inversely related to the time until the value will next be used. This interval function is [0,1]. The same weighting function is naturally extended to any (n,m) or [n,m] interval.

It is obviously impractical to perform such a calculation for all memory locations at all times.

It is sufficient, however, to consider those locations that are "live" or "active" at each point in time. Detection of such active periods of memory locations (M-lives) can be done in a way much similar to the detection of register lives. Some number K must be selected as the maximal dormant period permitted within an M-life. This corresponds roughly to an interval function of type (K,K). Since every location must be referenced at least every Kth instruction in order to stay live, at most K locations can be live simultaneously. A K chosen for this purpose would hardly be larger than 256. Hence the data space required for detection of M-lives is definitely manageable. A hashing scheme must be used to access the tables of M-life data, rather than the register address that was used for the R-life tables. Finally we must keep track of values that migrate from memory to registers and back.

An appropriate weighting function would probably take into account only future usages of the location. By using a lookahead of K instructions, the utilities of the live memory locations could be calculated.

We did not do this, but propose it as a possible tool to use for assessing the utility of a larger register block, or to assess the optimal size of a register block assuming a future more intelligent compiler.

4.8 Register structure, Conclusions

We now conclude the presentation of our methods for register structures. We have shown how to detect register lives, how to find the number of simultaneous lives and how to find an upper bound on the time cost incurred if the number of registers were to be reduced. Our results are summarized in sections 4.4.1, 4.5.1 and 4.6.5. On the whole, our experimental results seem to indicate that the time cost incurred by having only 8 general registers on the PDP-10 would not be excessive. (This assumes that instruction word space was needed for other purposes).

This number depends, of course, on other architectural properties of the ISP. If the registers were specialized, or if base registers were introduced, a larger number of registers would be needed. This is clearly seen in the results of Alexander [AleW72], 4 or more registers in the IBM 360 were kept busy as base registers. On the other hand, if the registers were removed from the address space and no register to register operations were introduced, memory would have to be used for temporaries, and fewer registers would be needed.

It should also be noted that the results for a reduced register block, though they are upper bounds in one sense, can not be attained unless the register allocation policy of the compilers is sufficiently clever. In particular, dormant periods should be recognized, and no registers should be allocated to a fixed purpose.

Finally we point out that a reduction in the number of registers, or a specialization of them, is likely to imply a higher programming cost, since the programmer will have to spend more thought to how he allocates them.

On the whole, register usage is determined more by the language and its implementation than by the algorithm. This is not surprising, since the programmer usually has no control over register usage. The observation is particularly true for languages that use a run time system, or otherwise impose a strong regimen on the structure of their object code. Thus our ALGOL and BASIC programs distinguish themselves in most of the results in this chapter, whereas systematic register use by BLISS and FORTRAN is lacking.

We have also presented a method for classifying register lives with the object of assessing the need for generality of registers. Again our results indicate that register generality is not extremely beneficial to program efficiency, and that little would be lost if the PDP-10 had, say, 2 floating point accumulators, 2 fixed point accumulators and 8 index registers. However, the other motivations for general registers have not been invalidated.

CHAPTER 5

DATA TYPES AND OPERATORS

We now turn to the data types of the processor, and the operators to manipulate data of these types. We look at two problems:

a) How to detect types and operators that are in the ISP, but are not sufficiently used to justify their inclusion. This is done by frequency counts and various derivatives thereof, as described in Section 5.1.

b) How to detect data types and operators that are not in the ISP, but could be included at a benefit. This problem may be approached by studying instruction sequences and operand values. We discuss this in Section 5.2 through Section 5.5.

Again, we will be mostly concerned with the time cost. Most of the methods described in this section also apply to control operators and in part to address calculation methods, as will be further discussed in Chapter 6 and Chapter 7. As an introduction we give some general comments on data types and the associated costs.

A <u>data type</u> is an interpretation rule which assigns meaning to the contents of one (or more) word(s), or parts of words. A data type is present in a computer if there are instructions that manipulate it. We list some commonly occuring data types and in some cases the associated operations or other characteristics.

Word (LOAD, STORE)
Arithmetic (Test of magnitude or sign)
Integer (Single, multiple or variable length)
Floating point (Single, multiple or variable length)
Address (LOAD, STORE)
Bit (Test, set)
Bit vector (One word, logical and other operators)
Character (Including 8-bit bytes as in the IBM 360 etc.)
Character string
Byte (Variable-length bit string or field)
Byte string

Byte pointer (Generalized address)
Word vector
Vector
Matrix
Array
List
Stack
Stack pointer
Instruction (Execution)

This list is not exhaustive, and the types listed are neither well defined nor disjoint. Some exist only for transfer purposes, the data operations being subsumed under some other type. Some are generalizations of others, i.e. the PDP-10 byte and byte pointer types generalize all partial word transfer operations (Address, bit, character, character string etc). The variable length arithmetic types will usually only exist on character or decimal based machines, i.e. business oriented machines.

The cost of including a data type in an ISP has several components:
Consumption of space for the opcodes in the instruction word.
Cost of hardware to implement it.
Possibly longer time to decode the whole instruction set.
A data type included in the ISP should be used sufficiently to warrant these costs, as discussed in Section 5.1.

On the other hand, a data type or some of its operators might not be present in the ISP although it is much needed in applications. This usually means that the necessary data structures and operators have to be implemented (interpreted) in terms of the existing data types and their operators. The cost shows up as:
Increased execution time
Increased space for program
Increased time for programming
Possibly increased space for data
Less readable programs, implying an increased programming cost.
This is discussed further in Section 5.2 through Section 5.4.

A missing but desirable data type might also be a variant of an existing type where the existing type is used instead. Examples of such types might be short integers† or Booleans (i.e. true/false valued). Since such types are simulated by existing ones††, their desirability does not manifest itself as an instruction sequence. The costs of not having such data types are:

Space cost of unnecessarily occupied memory.

Time cost of using the slower instructions.

We discuss this further in Section 5.5.

5.1 Frequency counts

The obvious way to expose infrequently used data types and operators is to accumulate the number of executions of each instruction. This table of execution counts, the instruction frequency table or IFT, is another compact data base which may be stored and used at a later time to obtain additional information. For a given ISP, the IFT has a constant size, hardly more than 512 words for any ISP.

Once it is built, the IFT can be printed out sorted by opcode, frequency of execution, or time spent executing each instruction. From this we can immediately see which operators are little used and might be candidates for omission. Similarly, instructions and instruction groups where the fraction of time spent is significantly larger than the fraction of instruction executions, are possible candidates for improved implementation. A variant of the IFT (see below) is presented in Appendix D. In Figure 5-1 we tabulate the number of different opcodes used by each subject program, and in Figure 5-2 we tabulate how many different opcodes account for 75%, 90% and 99% of the executed instructions for each subject program.

Clearly one can not omit instructions from the ISP on the strength of their non usage by one program. Hence it is necessary to build IFTs that are the sum of IFTs for individual programs. Summation can be over the whole subject set, or a subset thereof. When computing such IFTs, the data for each program should probably be normalized to account for the different program lengths, and also possibly weighed to account for the importance of each subject program. We call such an IFT a SWIFT (Summed Weighed IFT).

- - - - - - -

† Partword loads and stores with fullword arithmetic is not in general sufficient because of conventions for representing negative numbers, and overflow warnings.

†† Fullword integers and bit vectors for short integers and Booleans.

Another form of summed IFTs is the SNIFT (Summed Normalized IFT); A SNIFT is reproduced in Appendix D, including the printouts sorted by instruction count and computed time, as well as the FGR function. It was computed by normalizing each subject program to one executed instruction, summing the resulting IFTs, and renormalizing to 1 million. This permitted the use of our existing program, using integer arithmetic, but caused a few rounding errors in the type conversions. Hence the total counts given by the program are sometimes a few instructions off the exact million. By scaling to a round number, the individual results are easily interpreted as fractions. The FGR function and other results from this total SNIFT, and the SNIFTs for the compiler set and the numeric and nonnumeric sets, are given at the bottom of the respective tables in this section. Since we did not weigh our programs, some instructions, particularly unrounded arithmetic, which are frequent in some special contexts in our short programs, received counts that seem unreasonably high.

FIGURE 5-1

Number of different opcodes used by subject set.

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	112	126	88	151	154
Crout	104	109	52	87	94
Treesort	100	95	33	58	73
PERT	109	109	60	126	129
Håvie	113	122	85	140	145
Ising	104	-	44	121	125
Secant	-	-	-	149	152

Algorithm\Programmer	E	B	A	G	L
Aitken	49	51	50	52	52

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	158	129	130	153	162

Total subject set:	274	Compiler set:	227
Numeric set:	239	Nonnumeric set:	211

FIGURE 5-2

Number of opcodes accounting for
75%, 90% and 99% of the executed instructions

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	75%	15	14	16	25	24
	90%	37	19	31	55	53
	99%	77	49	66	112	111
Crout	75%	22	13	7	11	12
	90%	34	19	14	21	19
	99%	60	39	28	47	37
Treesort	75%	5	14	5	5	6
	90%	8	19	8	8	9
	99%	28	30	21	21	24
PERT	75%	18	13	9	9	9
	90%	37	18	18	19	21
	99%	63	39	41	66	69
Håvie	75%	28	19	18	18	18
	90%	42	34	26	23	23
	99%	57	55	61	74	82
Ising	75%	22	-	8	9	9
	90%	35	-	15	19	23
	99%	58	-	33	61	75
Secant	75%	-	-	-	8	8
	90%	-	-	-	20	17
	99%	-	-	-	55	56

Algorithm\Programmer		E	B	A	G	L
Aitken	75%	11	12	10	8	7
	90%	21	22	18	14	12
	99%	37	40	38	35	34

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	75%	26	22	15	20	18
	90%	49	39	30	40	35
	99%	94	80	63	81	74

	75%	90%	99%
Total subject set:	29	67	133
Compiler set:	27	53	114
Numeric set:	28	60	129
Nonnumeric set:	17	44	103

For some of the above results, as for the computed time in general, individual instruction execution times are needed. They can be taken from the manual of the processor in question or other available sources. In some cases assumptions have to be made about the average properties of the operands. These assumptions may have critical importance in the case of variable length operands (including bytes) but should otherwise be of little consequence by the law of large numbers. If variable length operands are common, this source of error may be reduced by including in the trace sufficient information that the correct execution time can be computed during analysis.

Except for the possible dependence of instruction times on operands, tracing is too powerful a tool to obtain the IFT. A counter in each straight line piece of code in the subject program plus the necessary data on each such piece, or jump tracing, would be sufficient. Tracing does, however, have the advantage of general applicability as discussed in Chapter 1.

We now discuss some further measures computed from the IFT.

5.1.1 Instruction classification - Mixes

In order to better see the relation of the instruction executions to the data types and other programming structures, we may group our instructions into classes and print the distributions of instruction counts or computed time over the classes. The classification may be by data type, control function or other properties. In some cases several data types may be grouped into one class; in other cases a data type may be split into several classes etc., depending on the questions to be asked. This may be viewed as mapping the instruction set into a generalized and smaller instruction set.

Two such classes were used in our work. One of these was devised by Gibson [GibJ70] in 1959, and used to obtain the well known Gibson mix. It has later been modified to fit more modern computers by Gonter [GonR69] and the present author. This classification was intended mostly for comparison of the internal processing power of different central processors. Another classification, The Program Structure classification (or PS classification), was developed by the present author. It is intended to reflect the control operators of a program in a better way than does the Gibson classification. The definitions of these classifications are given briefly in Figure 5-3 and Figure 5-4. For the full definition of the Gibson classification we refer to the papers by Gibson and Gonter.

We use the term distribution (Gibson distribution, PS distribution) to denote the observed distributions for any (set of) program(s). By a mix we mean the observed distribution for a set of programs believed to be representative of some actual workload (i.e. the Gibson mix [GibJ70], the UMASS mix [GonR69] etc.).

A classification is easily described by a table with one entry for each instruction in a standard format, and with some further entries describing the number of classes etc., and giving their print names. This table can be interpreted by the program computing (and printing) the distribution over the classes and the same program can be used for all distributions.

The original Gibson mix for the IBM 650 and 704, the UMASS mix for the CDC 3600, and the Gibson distribution for our subject set from the PDP-10, are reproduced in Figure 5-3. Our program structure distribution for the subject set and its subsets is given in Figure 5-4. When studying such distributions one should keep in mind that the number of instructions in each class is not the same. Hence a class of a few instructions averagely used may have a low count compared to a class of many instructions that are little used.

5.1.2 The FGR function and similar measures

The most striking observation from a quick glance at an IFT is that a small number of instructions account for a large fraction of the executed instructions. An abbreviated form of our results is displayed in Figure 5-1 and Figure 5-2. This suggests that one might reduce the instruction set and set of data types at a low cost. Foster et. al. [FosC71a] have proposed two measures related to this, they were both defined in Section 1.4, but we repeat the definitions here.

One of their measures is the information-theoretic measure of information content:

$$I = -\sum_{i=1}^{T} p_i * \log2(p_i)$$

where

p_i is the probability of using the i'th opcode
T is the total number of different opcodes
log2 is the logarithm base 2

Their other measure is a function computed as follows: Order the operation codes by frequency of occurrence. The i'th opcode in this ordering occurs C_i times, i.e. $C_i \geq C_{i+1}$ for $1 \leq i \leq P-1$, where P is the total number of instructions in the sample. The FGR function is then defined as:

$$FGR(N) = 1 - 1/P \sum_{i=1}^{N} C_i \qquad (1 \leq N \leq T)$$

FGR(N) is that fraction of the instructions which would have to be interpreted, were the instruction set reduced to the N most frequent instructions. However, the function does not guarantee that the implied recoding is possible or feasible.

Both of these measures are easily computed from the IFT. They may be computed based on the number of executions of each instruction, i.e. using the instruction count, or based on the time spent executing each instruction, i.e. using the computed time. The exact instructions "removed" depend, of course, upon this choice. In the latter case, C_i should be the time used by the i'th instruction when the instructions are ordered by the time spent executing them. Both the information-theoretic measure and the FGR function may also be computed from static data, and will then measure cost of representation rather than cost of execution.

We have computed the information-theoretic measure with respect to both instruction count and computed time. Although the practical value of these measures is small, they give some indication of the overall utilisation of the instruction set. The results are tabulated in Figure 5-5.

A much better measure is the FGR function, which gives an estimate of the time cost incurred by reducing the instruction set. We compute this based on instruction count, and with a simple extension. Assuming that each of the omitted instructions can be recoded in terms of K of the N remaining instructions, one may easily compute the relative increase in instruction count. If the instructions used for the recoding are of average time, the relative increase in computed time will be the same as that in instruction count. The increase in space cost has to be found by static methods, the FGR function computed using static instruction counts gives the fraction of written instructions that have to be rewritten.

In Figure 5-6 we tabulate the extended FGR function for N=64, N=48 and N=32, assuming a recoding factor (K) of 4, i.e. on the average 4 instructions needed to interpret each omitted instruction. This factor is the most significant source of error and is very hard to estimate, since many of the infrequently executed instructions are such that would require many other instructions to mimic exactly, but they are used where minimal changes of a larger context would get the intended operation done at no or very little extra cost. Hence the choice of K should be based on which instructions are candidates for omission. If, for instance, the floating point instructions are in danger, a factor of 4 will certainly be too low.

Ideally one would want to compute these costs using actual recodings of each omitted instruction. This might also give some information on the possible increase in space cost for data. This process is, however, not easily mechanized. Manual recoding is time consuming, since for each N considered one must code the missing instructions in the most optimal way using the N remaining instructions. Possibly the data representation must also be reevaluated each time. The recoding may also depend on space and time constraints for the particular application.

To properly see the costs of removing data types, results similar to those from the FGR function should be computed by removing all instructions relevant to a data type rather than the least frequently used ones. The results of such a calculation can usually be predicted well by a glance at the Gibson or PS distribution in question. Also, we believe it may be more relevant in many cases to omit certain of the operations of the data type rather than the whole type.

5.1.3 Summary of frequency results

Our experimental results indicate that a small number of instructions, at most 28, account for 75% of the executed instructions for any one of our subject programs, and that 112 instructions suffice for 99% of the instruction executions for any one program. No program used more than 162 instructions. Assuming a recoding factor of 4, 30 of the 41 programs could be run on a processor with 64 instructions at an increase of less than 5% in the number of instruction executions. For 18 of the programs this increase is less than 2%, but in 3 cases it runs as high as 20% to 30%. (ALGOL, FORTEN Bairstow, FORFOR Bairstow).

The situation changes somewhat when we consider the need of the whole subject set. Based

FIGURE 5-3

The modified Gibson classification.

Percentage of executed instructions in the Gibson classes.
Percentage of time included for our subject set.

Machine:	650/704	3600	KA-10	
	Gibsons	UMASS	Our results	
Class	results	results	Icount	Time
Load, store	31.2	30.0	42.4	35.6
Fixpoint add subtract	6.1	1.2	12.4	10.2
Compares	3.8	1.2	-	-
Branches	16.6	38.3	28.2	19.0
Floating add subtract	6.9	0.5	4.9	8.5
Floating multiply	3.8	0.5	2.6	8.7
Floating divide	1.5	0.2	1.1	4.9
Fixpoint multiply	0.6	0.1	1.1	3.2
Fixpoint divide	0.2	0.1	0.5	2.4
Shifting	4.4	2.2	3.9	5.3
Logical	1.6	0.5	1.0	0.6
Miscellaneous	5.3	0.0	1.5	1.7
Indexing	18.0	13.4	-	-
Fullword	-	6.9	-	-
I/O control	-	0.0	0.1	0.0
Inter reg. transfer	-	5.0	-	-
Monitor communic.	-	-	0.0	0.0
User UUOs	-	-	0.3	0.0

The classes are not equally applicable to all ISPs, as indicated by dashes. This applies in particular to index register instructions.

In Gibsons original classification, use of indexing was counted as an extra instruction in the "Indexing" class; the "Compare" class consisted of the 3 way skips in the 704.

In the UMASS version of the Gibson classification, the "Compares" class consists of all the vector search operations, "Indexing" is all the index register instructions, "Fullword" is all the 48 bit instructions. The "Inter register transfer" class also includes other instructions that only manipulate processor state.

Gibsons results were obtained using mostly scientific programs, but some business data processing programs, coded in unspecified languages.

The UMASS results were obtained using assembly and FORTRAN coded programs, including the FORTRAN compiler and the assembler.

FIGURE 5-4

The program structure distribution, part 1.

Percentage of instruction executions in each class for the total subject set and its subsets.

Class	Compilers	Nonnumeric	Numeric	Total
Word to acc.	10.5	24.2	19.7	20.1
Word to memory	4.6	9.4	7.2	7.6
Immediate to acc.	3.4	4.5	4.1	4.1
Set to acc.	1.3	0.4	0.3	0.4
Set to memory	1.2	0.2	0.5	0.5
Partword to acc.	10.8	4.0	3.2	4.4
Acc. to partword	2.4	0.5	0.7	0.9
Block move	0.2	0.0	0.1	0.0
Set bits	0.9	0.6	0.8	0.7
Add or sub. 1	1.6	1.8	1.6	1.7
Fixp. add sub.	5.3	14.5	9.7	10.8
Fixp. mul. div.	0.4	1.2	2.1	1.6
Floating arith.	0.0	1.4	15.1	8.6
Shifts	1.0	4.6	4.1	3.9
Logic	2.1	0.7	0.9	1.0
I/O transfer	0.0	0.1	0.1	0.1
I/O administr.	0.0	0.0	0.0	0.0
Other monitor comm.	0.0	0.0	0.0	0.0
User UUO	0	0.5	0.3	0.3
Subr. jumps	5.1	2.5	2.7	2.9
Subr. returns	3.9	2.2	2.2	2.4
Stackptr. manip.	5.5	3.3	4.9	4.4
Test acc. vs. immediate	7.7	1.7	1.0	2.1
Test acc. vs. 0	2.5	1.8	2.1	2.0
Test acc. vs. memory	3.0	4.9	4.5	4.5
Test memory vs. 0	2.3	1.7	0.9	1.3
Bit tests	7.4	1.2	1.4	2.0
Status tests	0.1	0.0	0.4	0.2
Loop jumps	3.9	3.3	3.6	3.6
Uncond. jumps	12.7	8.2	5.8	7.4
No-ops	0.0	0.0	0.0	0.0
Executes	0.3	0.8	0.4	0.5
Miscellaneous	0.2	0.0	0.0	0.0

The "Set to acc." and "Set to mem." classes load their destination with all zeroes or all ones. The "Set bits" group set individual bits in a word.

The program structure distribution, part 2.

Percentage of computed time in each class for the total subject set and its subsets.

Class	Compilers	Nonnumeric	Numeric	Total
Word to acc.	9.4	22.1	13.1	15.3
Word to memory	4.4	9.1	5.1	6.1
Immediate to acc.	1.8	2.5	1.7	1.9
Set to acc.	0.7	0.2	0.1	0.2
Set to memory	1.1	0.2	0.3	0.3
Partword to acc.	17.2	4.7	2.8	4.8
Acc. to partword	5.4	0.8	0.8	1.3
Block move	2.9	0.5	0.6	0.8
Set bits	0.6	0.4	0.4	0.5
Add or sub. 1	1.7	1.9	1.1	1.4
Fixp. add sub.	5.1	14.4	6.8	8.8
Fixp. mul. div.	1.5	7.2	5.5	5.6
Floating arith.	0.1	3.4	34.5	22.1
Shifts	1.0	4.5	6.5	5.4
Logic	1.8	0.5	0.5	0.6
I/O transfer	0.0	0.0	0.0	0.0
I/O administr.	0.0	0.0	0.0	0.0
Other monitor comm.	0.0	0.0	0.0	0.0
User UUO	0	0.0	0.0	0.0
Subr. jumps	5.1	2.6	2.0	2.5
Subr. returns	4.6	2.6	1.9	2.4
Stackptr. manip.	8.2	5.1	5.4	5.6
Test acc. vs. immediate	5.0	1.1	0.5	1.1
Test acc. vs. 0	1.6	1.2	1.0	1.1
Test acc. vs. memory	3.0	5.0	3.4	3.8
Test memory vs. 0	2.2	1.6	0.6	1.1
Bit tests	5.3	0.1	0.8	1.3
Status tests	0.0	0.0	0.2	0.1
Loop jumps	2.9	2.4	1.8	2.1
Uncond. jumps	7.1	4.6	2.4	3.5
No-ops	0.0	0.0	0.0	0.0
Executes	0.1	0.4	0.2	0.2
Miscellaneous	0.2	0.0	0.0	0.0

FIGURE 5-5

Information theoretical measure of opcode utilization.
Computed based on instruction count (IC) and computed time (CT)
Theoretical maximum (all opcodes equally probable) is 8.7245

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	IC	4.64	4.49	4.85	5.38	5.37
	CT	4.52	4.63	4.65	5.00	4.83
Crout	IC	5.10	4.44	3.75	4.46	4.36
	CT	5.15	4.51	3.67	4.46	4.39
Treesort	IC	3.21	4.40	3.17	2.93	3.36
	CT	3.03	4.51	3.16	2.94	2.95
PERT	IC	4.91	4.39	3.93	4.13	4.14
	CT	4.89	4.46	3.98	4.21	4.24
Håvie	IC	5.46	4.89	4.94	4.86	4.91
	CT	5.36	4.85	4.66	4.34	4.31
Ising	IC	5.19	-	3.88	4.18	4.30
	CT	5.19	-	3.77	4.29	4.42
Secant	IC	-	-	-	4.08	4.04
	CT	-	-	-	4.08	3.95

Algorithm\Programmer		E	B	A	G	L
Aitken	IC	4.26	4.27	4.09	3.76	3.66
	CT	4.02	3.97	4.12	3.99	3.94

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	IC	5.44	5.37	4.84	5.20	5.01
	CT	5.48	5.20	4.73	5.29	5.08

	IC	CT
Total subject set:	5.48	5.63
Compiler set:	5.62	5.62
Numeric set:	5.50	5.44
Nonnumeric set:	4.81	4.92

FIGURE 5-6

The extended FGR function.
Relative increase in instruction count by reducing the instruction set to 64, 48 or 32 instructions, using a recoding factor of 4.

Algorithm\language		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	64	0.092	0.021	0.043	0.294	0.268
	48	0.225	0.042	0.140	0.496	0.461
	32	0.483	0.094	0.360	0.792	0.755
Crout	64	0.022	0.006	0	0.006	0.005
	48	0.134	0.016	0.001	0.032	0.017
	32	0.447	0.081	0.023	0.174	0.093
Treesort	64	0.003	0.001	0	0	0.000
	48	0.006	0.004	0	0.000	0.002
	32	0.026	0.018	0.000	0.003	0.007
PERT	64	0.027	0.004	0	0.042	0.051
	48	0.184	0.019	0.012	0.081	0.103
	32	0.249	0.069	0.098	0.167	0.203
Håvie	64	0.018	0.024	0.029	0.059	0.077
	48	0.222	0.060	0.010	0.115	0.128
	32	0.750	0.454	0.235	0.216	0.224
Ising	64	0.020	-	0	0.035	0.078
	48	0.100	-	0	0.073	0.163
	32	0.476	-	0.041	0.157	0.288
Secant	64	-	-	-	0.024	0.026
	48	-	-	-	0.060	0.058
	32	-	-	-	0.184	0.160

Algorithm\Programmer		E	B	A	G	L
Aitken	64	0	0	0	0	0
	48	0.000	0.000	0.000	0.000	0.000
	32	0.128	0.162	0.109	0.052	0.050

Source progr.\Compiler		ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	64	0.210	0.109	0.036	0.101	0.073
	48	0.406	0.253	0.121	0.273	0.197
	32	0.779	0.565	0.341	0.579	0.463

	128	64	48	32
Total subject set:	0.056	0.422	0.631	0.926
Compiler set:	0.019	0.271	0.462	0.807
Numeric set:	0.040	0.352	0.574	0.883
Nonnumeric set:	0.010	0.199	0.342	0.585

on the SNIFT, the total number of instructions used is 274. 29 of these are sufficient to account for 75% of the instruction executions, 133 of them cover 99% of the instruction executions. The increase in time cost for recoding in a 64 instruction set is 42.2%. This recoding cost is well above the highest costs for individual subject programs. This shows that altough each individual program uses only a small set of instructions, this set is not the same for all the programs. Recoding into an 128 instruction set would increase the time by 5.6%.

The results vary systematically with algorithm and language. BLISS programs generally use fewest opcodes, and have the lowest recoding cost. This may in part be due to the total lack of run time system in BLISS (no I/O initialization or timing unless explisitly requested). BLISS programs are also as fast as, or faster than, the other programs for the same algorithm. Except for Bairstow, ALGOL programs have the highest recoding cost for a 32 instruction set but the FORTRAN programs, except for Crout, are the most expensive to recode in a 64 instruction set. The recoding cost of SEC is comparatively low, whereas it is consistently high for the compilers, though not higher than for several of the short programs. Treesort has the lowest recoding cost in all languages, Bairstow has the highest, except in BASIC. Hence there seems to be a correlation between the recoding cost and the size and complexity of the program. This is as one would expect. The difference between the results from the two FORTRAN versions seems significantly less than the difference between the results for the different languages.

When removing an instruction from an existing ISP, one should not only consider its frequency of usage, but also the ease of coding it in the remaining instruction set, and the degree of system in the allocation of opcodes. A break in such a system may cause increased programming cost. This is particularly true for the PDP-10, which has a very systematic instruction set.

The restricted selection of our subject set, and our use of SNIFTs instead of SWIFTs, casts some doubt on our conclusions about the necessity of individual instructions in the PDP-10. In particular, since all programs weigh equally, instructions used in special contexts in one of the small programs will get high representations in the SNIFT. Furthermore, the omission of I/O from the small algorithms leaves a timeconsuming and specialized aspect of most programs uninvestigated. We do, however, give some indications based on the SNIFT, which intuitively seem relatively independent of these deficiencies.

Large sections of the logic instructions (only 6 out of 64 are used significantly), the bit test instructions (9 of 64) and the halfword instructions could be removed. The systematic allocation of opcodes would not be unduly broken, and few instructions would need interpretation. There are also unused sections of the loop control group and the arithmetic group.

The UUOs are particularly little used. Their number could probably be reduced to 7 (3 user + 4 monitor) or 15 (3+12) by encoding information about function in the address field or in a control block. UUOs are further discussed in Section 6.1, where the time cost of using them is shown to be high relative to using routine call instructions.

Finally there are many no-ops and duplicate instructions. Removal of these would, however, break the systematic allocation of operations.

These remarks indicate that these results depend more on the algorithms than did those for registers. Hence a subject set should be chosen to cover the application area in the widest possible way. It should further contain as wide as possible a range of programming constructs. Commonly used languages should also be well represented. Finally one should not put too much significance into the results from one or a few analyses, particularly not from a small program.

We finally point out that the Gibson and program structure distributions (Figure 5-3 and Figure 5-4) indicate that there is also a great deal of commonality between the results from the different programs, and also between different ISPs.

5.2 Collection of instruction sequences

We now turn to the problem of detecting data types and operators that might be added to the ISP with benefit, and which represent data operations genuinely different from the existing ones. As previously noted, one way of detecting such operators may be by observing frequently occurring sequences of instructions, viz. those sequences used to perform the data operations, representing encodings of the missing instructions in terms of the existing instruction set.

5.2.1 The program

We first describe our method for detecting frequently occurring sequences of instructions. The major problems are due to the need for space and time efficiency in the analysis program. This is clearly demonstrated by a glance at the intermediate results of a large analysis[†]: 1600 different pairs were found by our program[††]. If all of these were to be extended to triples, quadruples etc., data space and processing time requirements would soon become prohibitive. Hence some methods are needed to detect and omit insignificant sequences.

The data structure where the information is collected is essentially a forest of binary trees [KnuD69], each node represents a sequence, and each root corresponds to the first instruction of the sequences represented in its tree. By a level (or level L) we mean all nodes representing sequences of a given length L. The leader of a sequence of length L is the L-1 first instructions in it. Its trailer is its L-1 last instructions. The descendants of each node are:

a) The extension, i.e. the first of the nodes on the next higher *level*, representing an extension of the sequence represented by this node.

b) The next, i.e. the next node on the same *level* having the same *leader*.

To facilitate pruning, as described below, we also chain all nodes on the same *level*, and in order that we may reconstruct the sequence represented by a node, each node has a back pointer to the node representing its *leader*. Finally each node contains the last opcode of the sequence it represents, the occurrence count for that sequence, and its length (i.e. the *level* number of the node).

For efficiency reasons we do not pack the nodes, hence 7 words are needed for each[†††]. 2000 nodes were sufficient for the analysis of all the subject programs except FORTEN. About 2100 nodes were needed for the first pass of that analysis, the 1600 mentioned on page 103 plus 512 for level 1.

- - - - - - -

[†] FORTEN, 295 000 instructions traced.

[††] Which were reduced to 61 after applying the pruning methods to be described.

[†††] Easily reduced to 4 words per node if using a language that makes the halfword load and store instructions available.

To keep the forest of limited acreage, we use a multi pass algorithm. The first pass accumulates the pairs, each subsequent pass extends the sequences by one instruction, thus adding one *level* to the forest. After each pass the forest is pruned. The pruning not only discards insignificant sequences, but also attempts to recognize closed loops, several representations of the same sequence, etc. If significant sequences remain after pruning, a new pass will be performed.

This continues until either all sequences on the top *level* are pruned or until a predetermined *level* (read as data) is reached. In the latter case, the user of the program may decide after each pass whether to continue. His decision is based on a few simple data typed as each pass is completed. Furthermore the current version of our program saves status after each pass and is easily restarted if inspection of the output indicates that longer sequences would be of interest, or in case of machine breakdown.

Maximal program capacity is sequences of length 20. This limit was arbitrarily set since we believed that sequences of this length neither would be found, nor would be of interest. This turned out to be only partly true. Using the pruning algorithm outlined below, and cutting each tree at the root when all its nodes at the top *level* are deleted, the algorithm is not prohibitively expensive†. Hence in the experiments we used a typed in limit of 20. About half of the analyses reached this *level*, all of them reached *level* 10.

After about the tenth pass of our algorithm very few sequences remain, hence each could probably be extended by 5 or more in each pass without undue consumption of space. This would make the method significantly faster, and permit the analysis to run until all sequences terminated "naturally". It would, however, require some reprogramming

At the end of the run the counts of shorter sequences are reduced to account for the extension of these sequences into longer significant sequences. That is: starting at the top *level* we visit each sequence in turn: and generate all its subsequences. For each such subsequence we reduce its count by the count of the main sequence. Hence the final count for each sequence reflects the unextendable fraction of the total number of occurrences of this sequence. The computed time for each occurence of the sequence is easily obtained, as are the fractions of the total instruction count and computed time consumed by all occurrences of the sequence.

----

† With approximately 100 000 instructions traced, (subject program FORTEN Treesort), the run time was approximately 35 min. for sequences of length up to 20. Probably this could be reduced considerably by coding the tree lookup routine in assembly code.

5.2.2 The pruning heuristics

The results presented in Section 5.3 were obtained using the following pruning algorithm: After each *level* is built, each of the new nodes is examined in turn and the heuristics about to be described are applied to it. Since some of the heuristics involve more nodes than the one thus examined, no nodes are deleted until a second pass down the *level* chain. The first pass merely marks the nodes to be deleted, using the extension field which is otherwise unused at the top *level*.

In the examples below, A, B, ... denote instructions, J denotes a jump instruction. A sequence and its count (the latter often omitted) are given as: <A B C D E: 647>.

Rule K:
All sequences whose count is less than 10% of the maximum count at the same *level* are marked for deletion.

Heuristic 0:
All sequences that are not a "significant" extension of their *leader* or *trailer* are marked for deletion. Exceptions are made for sequences of all the same instruction and for sequences whose count is at least 1/50 of the number of instructions in the subject program. The meaning of "significant" depends on the *level*. A factor is defined by the following table:

Level:	2	3	4	>4
Factor:	1/8	1/4	1/2	3/4

All sequences whose count is not at least factor times the count of both its *leader* and its *trailer* are marked. (If the *trailer* does not exist, its count is taken to be 0). The intent of this heuristic is to isolate the common part of partly overlapping sequences as the more important. Given the sequences <A B C: 500>, <B C D: 150>, <C D E: 150> and <D E F: 800>, <B C D> would not be marked, but <C D E> would be.

Heuristic 1:
The intent of this heuristic is to detect loops. It is applied at *levels* ≥ 4. It is first checked whether the first and last pairs of instructions in the sequence are the same. If so, it is checked whether the sequence contains a jump instruction. If so, we assume we have found a loop of length 2 less than the present *level*. Finally it is checked if the

same loop is represented elsewhere in the forest[†]. Whenever such a representation is detected, it is marked for removal. Thus <A B C D E F A B> and <A B C D J E F G> are not loops by this heuristic, but <A B C D J E A B> is a loop.

Heuristic 2:

This heuristic is applied at *levels* > 4. It attempts to detect if there are several nodes representing subsequences of the same longer sequence yet to be built. As the top *level* nodes are examined, chains are built linking nodes that are believed to represent such sequences. Let

S1 = <C D E . . . F G>

be the sequence of length L that is currently under examination. We now examine all sequences of form:

<X C D E . . . F >

for some X. Let S2 be one of these. S2 and its chain become the chain of S1 if:

a) Their count differ by at most 3.

b) S1 was not in this chain before.

a) will ensure that the sequences are equally significant; b) that we do not delete all representations of a loop. Note that S1 occurred later in the instruction stream than S2, but is before it in the chain. Hence the sequence occurring earliest in the instruction stream is the one which will have a null link, and therefore be kept. Thus for the sequences of <A B C D E>, <B C D E F> and <C D E F G>, the chain would go from <C D E F G> to <B C D E F> to <A B C D E>, and the latter would be kept. In the previous notation, if the chain consisted of S1 and S2, S1 would be deleted.

Heuristic 3:

This heuristic is applied at *levels* > 6, and is designed to detect and mark all but the most frequent of those sequences at the *level* which overlap by a significant number of instructions, - at least 2/3 of the level number. For each sequence at *level* L > 6 (say <A B C D E F G H>), we consider all extensions of its *trailer* to the *level* of L (such as <B C D E F G H I>), and delete all but the one with the largest count. We then repeat the process for the *trailer* of the *trailer* (i.e. <C D E F G H>), extending to *level* L again and so on until we have reached the least overlap permitted.

Each of these heuristics is programmed as a routine, and called from one place in the

- - - - - - -

† A loop of length L may be represented at L places in level L+2, each starting with a different instruction of the loop.

program, inside a pruning control routine. Hence it is easy to change the heuristics and the order in which they are applied, or to add new heuristics.

5.2.3 Sources of errors

There are some problems associated with this method. Some of these could be avoided by adjusting the parameters to the heuristics, but this is not sufficient. We now present the most significant of these problems, and propose some remedies.

Sequence overlap

Because of the heuristic nature of the pruning algorithm, we have no guarantee that the sequences at any *level* are really disjoint. Hence the final reduced counts are not completely reliable. In particular the counts for subsequences common to two overlapping longer sequences will be too low. This is clearly seen in all programs analyzed, several examples are shown in Section 5.3.

To remove this problem, the heuristics for detecting overlaps must be improved. At first sight, the obvious way is to shift each sequence completely out of the sequence detection mechanism once it has been recorded, rather than trying to detect new sequences starting with instructions in its *trailer*. This assumes, however, that the sequence just recorded is more significant than those omitted as a consequence of the shift. Hence this technique can not be used at low *levels*, since that would prevent us from detecting which sequences are significant in the first place. Changing to this technique at a higher *level* requires great care lest we extend the wrong sequences of those now overlapping. Hence we reject this approach, and we believe the way to go must be to improve our present heuristics and the way they interact, and device new heuristics in the same spirit.

We believe that not even the best of heuristics can completely avoid this problem. Hence we suggest two more ways to relieve it. Firstly, the counts at each *level* may be printed after the *level* is built, immmediately before pruning, as well as at the end of the analysis. These <u>original</u> counts may then be compared with the final <u>reduced</u> counts. We did this, and found it a help in detecting significant sequences in general during the manual analysis described in Section 5.3. In Section 5.3 we present both original and reduced results.

Secondly, one may decide from one run as outlined above, which sequences are important enough or which results are wrong enough that exact counts are desirable. A second run can then be done, with a slightly different program, collecting statistics on these sequences only. This can be done in one pass since we know what to look for. Such a program should be written to look for classes of sequences as, for instance, variants of a calling sequence, possibly defined by a regular expression. We wrote no program for this.

Dominating loops

Another problem is that of dominating loops. Our program tends to find long sequences, sometimes representing whole loops of the subject program, rather than the shorter sequences that are more frequent and which could reasonably be implemented as instructions. This is particularly true for the short subject programs, where one or a few loops dominate the results. The situation is improved when subject programs of a more representative length and complexity are analyzed. Further improvement can most probably be achieved by strengthening the definition of "significant" in heuristic 0. This can be done either by increasing the "factor", particularly for the higher *levels*, or we may introduce new criteria of "significance". One such could be to compare the total time consumed by the sequences in question rather than their occurrence counts. Again a factor could be used in a way similar to the present one.

Interacting heuristics

A third problem is the interaction of the heuristics, particularly heuristics 1 (loops) and 2 (subsequences of longer sequences). Probably the loop heuristic should be applied last, after all deletions resulting from the other heuristics have been performed.

Semantics of sequences

Finally there is the problem of relating the sequences back to the subject program in question. This may be difficult because the semantics of the sequences is not always obvious, and can only be found after a careful and time consuming study of well commented source and assembly listings. Also, the sequences found may not relate easily to intuitively meaningful notions. This is related to the problem of dominating loops. The double length arithmetic of Crout is a case in point. This occurs in a context such as

<u>for</u> kx ← low <u>step</u> 1 <u>until</u> high <u>do</u> sum ← sum+A[lx,kx]*B[kx];

where sum is the double length variable. The double length addition is easily spotted by the occurrence of the UFA† instruction, but it is embedded in a sequence of length 20 which also involves array accessing and the enclosing loop.

More intuitive program elements can be brought out by:

Looking for more specific sequences as indicated above.

Improving the heuristics, possibly to start and break sequences at jumps more easily than now. However, an advantage of our present method is that it permits detection of significant sequences, crossing transfers of control, that might not have been suspected to be of importance††. This property should not be lost.

Generate sequences longer than 20, and try to keep the "earliest" one as described under heuristic 2.

5.3 Results from the sequence program

Each result produced by our program consists of a sequence of operation codes, together with its occurrence count and timing data computed from this count. Hence the results need quite a bit of manual analysis to yield useful data. This analysis involves comparing with assembly listings (possibly using interactive debugging systems to locate sequences), comparing counts obtained before and after reduction or on different *levels*, etc. Good knowledge of the subject program in question is an obvious advantage.

The deficiencies of our pruning heuristics and the way they interact, as described in Section 5.2.3, increase the difficulty of this analysis. We have, however, made an attempt, and present the results below. Due to the manual processing, the selection of sequences presented is necessarily subjective.

- - - - - - -

† Unnormalized floating add

†† The BLISS calling sequences, the array access and UUO handling in BASIC programs, and the thunk of ALGOL PERT are examples of this.

The results are presented by algorithm. The characteristics of each algorithm, as described in Figure 3-2, rarely occur frequently enough to show up, but when they do we comment on it. For each program, the maximal sequence length reached during analysis is given. In some cases all sequences on the highest *level* reached were deleted by the pruning mechanism. In those cases the highest *level* with significant sequences was one or two lower than the highest *level* reached, as is indicated in parentheses. In some cases the sequences at the top *level(s)* were rejected during the manual scan. This is not explisitly indicated.

Since this method of sequences is applicable to address calculation and control structures as well as to data types and their operators, we have made no distinction between sequences of these 3 types in the lists of sequences. For the same reason we present them with the bare minimum of identifying comment. Evaluation is postponed until later sections in the relevant chapters: 5.4, 6.1 and 7.1.1.

The sequences are presented in a standard format, giving the occurrence count of the sequence, the percentage of the total computed time consumed by it, and a single letter (B or A) designating if the results are from before or after count reduction. This is followed by the sequence itself. Several versions of the same or largely overlapping sequences have been included when it seemed to be of interest, either because of a much larger count for a subsequence, because of a better correspondence with an intuitive program fragment, to show the difference due to count reduction, or to show examples of bad pruning. Since the sequences overlap, the percentages of time sometimes add up to more than 100.

Note that an XCT instruction is immediately followed by its target instruction. User UUOs† are given in numeric (octal) form, followed by the code for the UUO interpreter, starting at location 41. Monitor UUOs are given in their octal form, followed by the next instruction of the program itself (see Section 1.3).

----

† A user UUO is an instruction (octal 01 through 37) which causes a trap to location 41 in the users memory. Since the subroutine thus called is user defined, the UUOs do not have common mnemonic names. Monitor UUOs (octal 40 through 77) cause a trap to absolute location 41 and are used for monitor calls.

5.3.1 The compilers

Since these programs are large and complex, and little known to the present author, the analysis of them is in some cases less thorough than desirable. This applies in particular to the two FORTRAN compilers. In the other cases experts were available for consultation and the results of the analysis are better.

ALGOL

Maximal sequence length: 11.

Seq.	Count	% Time	B/A	Sequence
(1)	170	2.9	A	JRST LDB MOVE CAIN CAIE JRST JSP ILDB AOS MOVE XCT
(2)	117	2.0	A	LDB SKIPE MOVE IBP AOS POPJ JRST MOVEM MOVE MOVE MOVEM
(3)	115	2.0	A	MOVE MOVEI XORB MOVEM MOVEM PUSHJ SKIPE LDB SKIPE MOVE IBP
(4)	216	3.5	A	CAME POPJ IMULI ADDI SOJG PUSHJ ILDB AOS
(5)	295	2.8	A	JRST CAIN ILDB AOS MOVE JRST
(6)	333	3.5	B	AOBJN LSHC ILDB AOS SKIPL
(7)	541	5.6	A	PUSHJ ILDB AOS CAME POPJ
(8)	1641	9.3	B	ILDB AOS
(9)	176	2.4	A	PUSHJ ANDI MOVE HRRM MOVE MOVEM AOS CAME JRST HRLI MOVEM
(10)	109	2.2	A	MOVE PUSHJ TRNN POPJ MOVE MOVE ADDI ANDI IDIVI ADDI TLNN
(11)	1442	2.5	B	TLNE JRST
(12)	1418	3.7	B	MOVE MOVEM
(13)	917	2.7	B	AOS CAME

Sequences (1) to (8) represent various forms of input of characters. (9) and (10) are concerned with outputting relocatable code. (11) shows the need for test bit(s) and jump, (12) may be a memory to memory move, (13) is loop control.

BASIC

Maximal sequence length: 17.

Seq.	Count	% Time	B/A	Sequence
(1)	1104	20.7	A	SKIPE ILDB JRST CAIE CAIN CAIN CAIE CAIN CAIE CAIN CAIG CAIA CAIGE IDPB SKIPE SOSLE AOJA
(2)	990	8.0	A	ILDB CAIN IDPB JRST
(3)	456	2.9	A	ILDB HLL TRNE TLNE JRST
(4)	402	2.1	A	HLL TRNE HRL TLNE POPJ
(5)	517	5.7	B	ILDB HLL TRNE TLNE POPJ
(6)	521	2.9	A	PUSHJ ILDB HLL
(7)	314	3.3	A	MOVEI PUSHJ MOVE ADD CAIE SKIPA CAMLE EXCH POPJ MOVEM
(8)	677	3.5	A	CAIGE JRST MOVEI ADD ASH CAMLE

(1) Is a loop to move text lines from the TTY input buffer to the BASIC line buffer, character by character. As the line is moved special characters, like VERTICAL TAB, LINE FEED, RETURN. are removed or special action is taken on them. This loop could probably be reduced to two instructions (ILDB JRST) at the space cost of a one word table entry per character in the character set.

Sequence (2) represents the loop that moves a line from the line buffer into the program text area, stopping at a return. Further sequences, (3) to (6), are associated with the routine that reads the next character, sets appropriate flags depending on its properties, and ignores blanks.

The main data structure of BASIC is the <u>roll</u>, which essentially is a contiguous but dynamically relocatable memory area. The compiler has a fixed number of rolls, which are packed to conserve space and occasionally have to be relocated in order to let one of them expand. The sequences (7) and (8) relate to this data structure. The first of these adds a data item to the end of a roll, first checking if there is room. The second loop performs binary search in an ordered roll.

BLISS

Maximal sequence length: 10 (8).

Seq.	Count	% Time	B/A	Sequence				
(1)	15763	14.3	A	PUSH	PUSHJ	JSP	PUSH	HRRZ
(2)	10462	7.2	A	JRST	POP	POPJ	SUB	
(3)	3724	3.5	A	JRST	POP	POP	POPJ	SUB
(4)	4897	3.5	A	PUSH	HRRZ	PUSH	JRST	
(5)	4489	3.0	A	PUSH	PUSH	PUSHJ		
(6)	3264	2.4	A	PUSH	PUSH	PUSH		
(7)	18275	12.1	B	PUSHJ	JSP	PUSH	HRRZ	
(8)	12256	6.9	B	JSP	PUSH	HRRZ	JRST	

All these represent the routine entry and exit mechanism, which probably accounts for at least 25% of the compilation time. Note that these sequences have considerable overlap, and that (7) and (8) are from before reduction.

FORFOR

Maximal sequence length: 10 (8).

Seq.	Count	% Time	B/A	Sequence							
(1)	17484	11.3	A	AOJA	MOVE	HLRZ	TRNN	JRST			
(2)	14555	9.9	A	AOJA	MOVE	HLRZ	TRNN	TRZE			
(3)	6390	5.9	A	HLRZ	TRNN	TRZE	JUMPN	TRZE	AOJA	MOVE	
(4)	5750	7.0	A	HLRZ	CAIN	ADD	HRRZM	HRRZ	ADD	HRRZM	SOJE
(5)	4411	5.1	A	PUSHJ	LDB	ANDI	MOVEI	HLRZ	CAIG		
(6)	5635	5.0	A	SOJGE	HLRZ	CAIN	ADD	HRRZM	HRRZ		
(7)	26907	5.9	B	TRNN	JRST						
(8)	38569	10.8	B	HLRZ	TRNN						

This compiler is highly interpretive, simulating a one or few register machine on the 16 register PDP-10. Sequences (1) to (3) are associated with the "instruction fetch" cycle of this interpreted machine.

(4) to (6) are associated with roll maintenance. We believe that a roll in FORFOR is approximately the same as in BASIC (see under BASIC above), but since no FORTRAN expert is avaliable, and the assembly listing is poorly commented, we have not been able to verify this.

Some further short sequences, (7) and (8), with large counts and time were spotted in the output from before count reduction. They clearly demonstrate the need for a test bit and jump instruction.

FORTEN

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	571	3.2	A	CAIG	POPJ	CAIE	JRST	CAIE	JRST	MOVE
				TRNE	CAIE	JRST	CAIN	AOS	CAMG	JRST
				PUSHJ	MOVE	CAMGE	JRST	AOS	MOVEM	
(2)	949	3.2	A	CAIG	POPJ	JUMPE	MOVE	TRNN	JRST	CAIE
				JRST	MOVE	TRNN	JRST	SETZ	POPJ	
(3)	4960	4.7	B	POP	POPJ					
(4)	2532	4.1	B	PUSHJ	JSP	PUSH	HRRZ	JRST		
(5)	2403	4.7	B	PUSHJ	JSP	PUSH	HRRZ	PUSH		
(6)	1936	5.5	A	PUSHJ	SOSG	CAIA	ILDB	MOVEI	CAIG	POPJ

(1) and (2) show the need for good testing instructions. (3) to (6) are from the BLISS routine entry and exit sequences (FORTEN is written in BLISS). From these results it is reasonable to assume that the routine call administration consumes at least 15% of the time in FORTEN. (6) represents reading a character from input, with some additional administration.

5.3.2 SEC

Most of the sequences of this program represent loops of considerable length. Usually several matrix accesses can be observed in each loop, but these are not brought out separately after count reduction.

FORFOR SEC

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	2987	11.9	A	CAMGE	AOJA	MOVE	MOVEI	IMUL	MOVE	ADD
				MOVE	ADD	FMPR	MOVE	ADD	MOVE	ADD
				FMPR	FADR	MOVEM	MOVE	MOVEI	IMUL	

(2)	2340	5.5	A	CAMGE	AOJA	MOVEI	IMUL	ADD	MOVE	MOVE
				ADD	FMPR	MOVE	ADD	FADRM		
(3)	2987	3.0	A	MOVEM	CAMGE	AOJA	MOVE	MOVEI	IMUL	
(4)	9390	9.6	A	MOVE	ADD	MOVE	ADD	FMPR		
(5)	8777	8.0	A	MOVEI	IMUL	MOVE	ADD	MOVE		
(6)	11072	8.7	A	MOVEI	IMUL	ADD	MOVE			
(7)	15364	14.0	B	ADD	MOVE	ADD	FMPR			
(8)	12499	11.2	B	MOVE	ADD	FMPR	MOVE			
(9)	20181	15.7	B	MOVE	ADD	FMPR				
(10)	22128	14.9	B	MOVEI	IMUL	ADD				

(1) and (2) are loops as mentioned, (3) to (5) are sections of such loops, with loop control and matrix access showing. The original count for (2) was 2980, and 7% of the time was consumed by it. The original time was 15.7% for (4), 10.7% for (5), 12.8% for (6). (6) is a load of a matrix element. (7) to (10) are original results. The MOVE ADD OPERATE sequence is access to formal vector, (10) is the matrix accessing sequence.

FORTEN SEC

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	2987	12.7	A	ADD	MOVN	FMPR	MOVE	FMPR	FADR	MOVEM
				MOVEI	IMUL	ADD	ADD	MOVE	MOVEM	ADDI
				AOJL	MOVEI	IMUL	MOVE	ADD	ADD	
(2)	2980	7.3	A	FADRM	ADDI	AOJL	MOVE	ADD	MOVE	ADD
				MOVEI	IMUL	ADD	MOVE	FMPR		
(3)	4760	8.0	A	MOVE	FMPR	FADR	MOVEM	MOVEI	IMUL	
(4)	5940	3.9	A	MOVEM	MOVE	MOVEM	MOVE	MOVEM		
(5)	21006	5.4	B	MOVE	MOVEM					
(6)	11523	6.1	A	MOVE	ADD	ADD	MOVE			
(7)	10562	9.0	A	MOVEI	IMUL	ADD	MOVE			
(8)	34831	20.2	B	MOVEI	IMUL					
(9)	26790	19.4	B	MOVEI	IMUL	ADD				
(10)	7758	8.4	A	FMPR	FADRM	ADDI	AOJL			
(11)	5134	5.7	A	MOVE	FMPR	FADRM	ADDI			
(12)	12337	10.3	A	ADD	MOVE	FMPR				
(13)	22689	15.7	B	MOVE	FMPR					

Sequence (1) here is obviously the same loop as (1) under SEC40. (2) to (4) represent similar structures. The latter may indicate the need for a memory to memory move, as illustrated further by (5). (6) contains vector access. (7) is matrix element load. The importance of the matrix data structure is further illustrated by (8) and (9), from before reduction. (10) to (12) are of doubtful origin. (10) and (11) might represent some inner product like loop, (12) consumed 12.8% of the time using the values from before reduction. (13) would be considerably more efficiently executed on a two address design. The MOVE ADD OPERATE sequence represents the use of a formal vector and is present in several of the sequences.

5.3.3 Aitken

This algorithm consists of two phases, first a search in the vector of abscissae to locate the interval where interpolation is to take place, then the interpolation itself which is somewhat similar to successive calculations of two by two determinants, controlled by two nested loops. Depending on implementation the local data are a two dimensional array or some number of vectors. Also some implementations work directly on the parameter vectors defining the abscissae and ordinates, others move the values needed to local vectors thereby saving accessing code. Two implementations perform arithmetic on the values while so moved. All these variations show up clearly in the results to be presented.

The surrounding program, which sets up the vectors of function (logarithm) values, and calls AITKEN with different parameters, does not show up in the results from the most time consuming implementations of Aitken, but is very conspicuous in the results from the more efficient versions.

Aitken - E

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	200	8.2	A	FAD	MOVE	FAD	FDV	MOVE	FMP	MOVE
				FMP	FAD	FMP	FAD			
(2)	200	11.9	A	MOVE	FMP	MOVE	FMP	FAD	FMP	FAD
				FMP	FAD	MOVE	FMPR	JRST	POP	POP
				POP	POP	POP	POPJ	SUB	MOVEM	

(3)	198	6.1	A	CAMG JRST AOJA CAILE MOVE MOVEM JUMPLE MOVE CAML PUSH PUSHJ JSP PUSH HRRZ PUSH PUSH PUSH PUSH JRST MOVE
(4)	196	6.7	A	MOVEM MOVE FADRB MOVE FMPRB GAMG JRST MOVE CAMG JRST AOJA CAILE MOVE MOVEM JUMPLE MOVE CAML PUSH PUSH JSP
(5)	1485	49.4	A	MOVE FMPR MOVE FMPR FSBR MOVE FSBR FDVR MOVEM SOJGE
(6)	405	8.7	A	MOVE SOJ JUMPL MOVE FMPR MOVE FMPR FSBR
(7)	324	4.7	A	MOVEM SOJGE SOJG MOVE SOJ JUMPL MOVE FMPR
(8)	255	3.2	A	ASH CAML JRST MOVE JRST MOVE AOJ GAMG MOVE ADD
(9)	405	5.4	A	MOVE MOVEM FSBR MOVEM MOVE MOVEM AOJ AOJ SOJGE

Sequences (1) to (4) are from the controlling program, and represent the internals of LOG, its entry and exit, and the controlling loop. The two first and the two last overlap. As is seen, the routine entry and exit sequences are dominant, particularly the saving and restoring of local registers. There is also some indication of use of Horners rule.

Sequences (5) to (7) represent the determinant like loop, with the first being the inner loop, the next two the outer loop and partly overlapping the inner. Binary search in the abscissae vector is represented by (8), and vector move by (9). The original result for (9) was 6.4% of the computed time. Addresses of the vector elements are used directly in the code, to save address calculation.

Aitken - B

Maximal sequence length: 14 (12).

Seq.	Count	% Time	B/A	Sequence
(1)	1485	53.8	A	MOVE FSBR FMPR MOVE FSBR FMPR FSBR MOVE FSBR FDVR MOVEM SOJGE
(2)	405	6.9	A	MOVE SOJ JUMPL MOVE FSBR FMPR MOVE FSBR
(3)	324	3.4	A	MOVEM SOJG SOJG MOVE SOJ JUMPL MOVE FSBR
(4)	630	6.4	A	MOVE SUB CAIG MOVE ADD ASH MOVE CAML
(5)	405	3.3	A	AOJ AOJ SOJGE MOVE MOVEM MOVE MOVEM

(6)	282	3.9	A	POP	POP	POP	POP	POP	POPJ	SUB
(7)	444	3.7	A	PUSH	PUSH	PUSH	PUSH			
(8)	400	6.4	A	FMP	FAD	FMP	FAD			

The routine uses the addresses of the formal vectors directly, hence there is no extra accessing code. The determinant loop, and the partly overlapping sequences from its enclosing loop are almost as in the E version, as seen in (1) to (3). The binary search shows up as (4). The vector move of formal to local is (5), its original time was 3.8%. Procedure entry and exit is shown by (6) and (7). From the initialization we have (8), which is Horners rule in unrounded arithmetic.

Aitken - A

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	1320	38.4	A	CAMLE	MOVE	FMPR	MOVE	FMPR	FSBR	MOVE
				FSBR	FDVR	MOVEM	AOJA			
(2)	432	3.3	A	MOVE	AOJ	MOVE	SOJ	MOVEM	CAMLE	MOVE
(3)	288	11.0	A	CAMLE	JRST	AOJA	CAMLE	MOVE	AOJ	MOVE
				SOJ	MOVEM	CAMLE	MOVE	FMPR	MOVE	FMPR
				FSBR	MOVE	FSBR	FDVR	MOVEM	AOJA	
(4)	1920	8.6	A	MOVE	MOVEM	AOJA	CAMLE			
(5)	261	5.3	A	MOVE	CAMLE	SKIPA	MOVE	ADD	ASH	MOVE
				JRST	MOVE	SUB	CAIG	MOVE	ADD	MOVE
				CAME	JRST	MOVE	ADD			
(6)	540	7.4	A	MOVE	SUB	CAIG	MOVE	ADD	MOVE	CAME
				JRST	MOVE	ADD	MOVE	CAMLE		
(7)	360	7.2	A	CAMLE	AOJ	MOVE	ADD	MOVE	MOVEM	MOVE
				ADD	MOVE	MOVEM	MOVE	ADD	MOVE	FSBR
				MOVEM	AOJA					
(8)	282	3.5	A	POP	POP	POP	POP	POP	POPJ	SUB
(9)	400	7.2	A	FMP	FAD	FMP	FAD			
(10)	3433	7.3	B	AOJA	CAMLE					
(11)	3078	7.5	B	MOVE	ADD					
(12)	2538	9.1	B	MOVE	ADD	MOVE				

The determinant loop is represented by (1) to (3); the two latter represent the outer loop and also overlap the first, which is the inner loop. (4) is own to own vector move in the outer loop. From the binary search we have (5) and (6). The formal to local vector move is (7). The initialization phase shows up as routine exit and Horners rule, as shown by (8) and (9). (10) to (12) show the original results for loop control and access to formal vectors.

Aitken - G

Maximal sequence length: 14 (12).

Seq.	Count	% Time	B/A	Sequence
(1)	6336	41.9	A	MOVE ADD MOVE MOVEI CAML MOVE MOVEI CAMG AND TRNN AOS JRST
(2)	2970	17.0	A	MOVE ADD MOVE MOVE ADD FMPR
(3)	18837	23.5	B	MOVE ADD
(4)	11439	20.9	B	MOVE ADD MOVE
(5)	2970	11.5	B	MOVE ADD FMPR
(5)	1971	3.7	B	MOVE ADD MOVEM
(7)	1485	3.9	B	MOVE ADD FSBR

The search in the vector is linear, and represented by (1). The determinant loop is not represented significantly except for a short section which occurs twice in the loop and hence overrides the accumulation of longer sequences. This is (2), which represents multiplication of two vector elements. Other fractions of this loop are present but not significantly. The access to a local vector is of the format MOVE, ADD, OPERATE. This is shown in (3) to (7), from before reduction.

Aitken - L

Maximal sequence length: 18.

Seq.	Count	% Time	B/A	Sequence
(1)	1485	31.0	A	MOVE SOJ IMULI ADD MOVE FSBR FMPR MOVE FSBR ADD FMPR FSBR MOVE FSBR FDVR MOVEM AOJA CAMLE
(2)	1485	17.0	A	MOVE IMULI MOVE ADD MOVE SOJ IMULI ADD MOVE FSBR FMPR
(3)	6264	40.5	A	CAMLE MOVE ADD MOVE CAME JRST MOVE ADD MOVE CAMGE JRST AOJA
(4)	9127	9.5	B	AOJA CAMLE
(5)	15219	18.1	B	MOVE ADD
(6)	14247	24.8	B	MOVE ADD MOVE
(7)	1971	7.1	B	MOVE IMULI MOVE ADD

The sequences (1) and (2) represent the determinant loop. The vector search (linear) is shown by (3). The original results representing loop control and vector access are shown in sequences (4) to (6). (7) represents access to a matrix.

5.3.4 The CALGO algorithms, initial remarks

Before presenting the result for the CALGO algorithms, we make some general remarks about the languages and their peculiarities: For matrix access the present ALGOL implementation uses Iliffe vectors, whereas the other systems use multiplicative methods.

In ALGOL programs a complicated run time system is used to implement the parameter mechanism (call by name), space allocation and block structure, and to check the legality of operations. This is particularly noticeable in routine calls and parameter access. The run time system sequences are easily detectable by the bit manipulating instructions they contain.

BASIC uses a similar run time system. User UUOs are used to call the routines of this system, this even holds for routines to do vector and matrix access. Furthermore all arithmetic is in floating point, so the indexes must be truncated to integers. The routine to do this also checks the result against the upper bound. The code to fetch and store vectors is the same except for one MOVEI at the beginning which loads a register with a MOVE, MOVEM or MOVNM instruction. This is XCT'd from that register at the end of the access routine. The code for matrix access overlaps that of vector access to a large extent.

5.3.5 Bairstow

ALGOL Bairstow

Maximal sequence length: 11 (10).

Seq.	Count	% Time	B/A	Sequence						
(1)	345	9.6	A	JRST	AOS	CAMLE	MOVE	ADD	MOVE	MOVE
				ADD	MOVE	FMPR				
(2)	1001	24.5	A	MOVE	ADD	MOVE	FMPR	FSBR	MOVE	ADD
(3)	535	11.7	A	MOVE	ADD	MOVE	MOVE	ADD	MOVE	FMPR
(4)	516	6.4	A	MOVE	ADD	MOVE	JRST	AOS	CAMLE	
(5)	470	6.0	A	ADD	MOVEM	MOVE	ADD	MOVE	MOVE	
(6)	518	5.8	A	FSBR	MOVE	ADD	MOVEM			
(7)	3085	19.5	B	MOVE	ADD	MOVE				
(8)	1025	6.6	B	MOVE	ADD	MOVEM				

(9)	4710	20.3	B	MOVE ADD
(10)	637	3.8	B	JRST AOS CAMLE

Sequence (1) to (6) show mainly vector access (MOVE ADD OPERATE) and loop control (JRST AOS CAMLE) with some other operations intermixed. The results for the vector access and loop control before reduction are given as (7) to (10).

BASIC Bairstow

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	3488	35.7	A	MOVEI MOVE HRRZ TRNN JRST PUSHJ MOVE AOS MOVE FAD TLZ CAMGE POPJ ADD ADD XCT MOVE POPJ
(2)	1138	10.2	A	MOVEI MOVE HRRZ TRNN JRST PUSHJ MOVE AOS MOVE FAD TLZ CAMGE POPJ ADD ADD XCT
(3)	1171	4.9	A	JSR JRST PUSH LDB JRST JRST
(4)	4626	9.7	B	MOVE FAD TLZ

Sequence (1) gives all of the code for vector fetch, except the initial MOVEI. (2) gives the same for vector store, but truncated at the XCT instruction. The counts are correct, as can be checked against the count for the appropriate UUOs. (3) is the general UUO handler. Its original count was 4659, representing 19.5% of the time. (4) represents the conversion of indices to fixed point.

BLISS Bairstow

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	90	5.1	A	TRNN JRST SKIPE PUSH PUSHJ JSP PUSH HRRZ JRST 051 SETZ JRST POP POPJ SUB JRST MOVEI SUB JRST POP
(2)	452	22.4	A	MOVE FMPR MOVE FSBR MOVE FMPR FSBR MOVEM
(3)	370	9.1	A	MOVE FMPR FADR MOVEM
(4)	329	7.8	A	MOVEM AOJA CAMLE MOVE FMPR
(5)	263	6.6	A	FSBR MOVEM MOVE FMPR
(6)	263	6.6	A	FMPR FSBR MOVEM MOVE
(7)	276	5.3	B	PUSH PUSHJ JSP PUSH HRRZ JRST
(8)	376	4.4	B	POP POPJ SUB

(9)	819	4.3	B	AOJA CAMLE

Sequence (1) and several overlapping sequences not listed represent output to TTY. (2) is synthetic division of a polynomial with a quadratic term. (3) is an expression of form D[j] ← D[j]+R*D[j-1]. (4) to (6) are various parts of the important loops. (7) and (8) represent routine calling overhead. (9) is loop control.

FORFOR Bairstow

Maximal sequence length: 18 (16).

Seq.	Count	% Time	B/A	Sequence
(1)	181	18.4	A	FMPR FADR MOVNM MOVE FMPR FSBR MOVE FMPR FADR MOVNM CAMGE AOJA MOVE FMPR FSBR MOVE
(2)	181	9.7	A	FADR MOVNM CAMGE AOJA MOVE FMPR FSBR MOVE FMPR
(3)	226	10.9	A	MOVE FMPR FSBR MOVE FMPR FADR MOVNM
(4)	148	8.3	A	FMPR FADR MOVEM MOVE FMPR FADR MOVEM CAMGE AOJA MOVE
(5)	492	2.6	B	CAMGE AOJA
(6)	859	19.2	B	MOVE FMPR FADR
(7)	581	13.1	B	MOVE FMPR FSBR

Sequence (1) is the full loop of the synthetic division. (2) and (3) are probably sections of this loop which remain thanks to bad pruning. (4) is the same as (3) in BLISS Bairstow, but the full loop. (5) is loop control, (6) and (7) are timeconsuming combinations of arithmetic operations.

FORTEN Bairstow

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	44	4.4	A	MOVEM MOVEI PUSHJ CAIA MOVE JUMPG CAMN ASHC ADDI MOVSM MOVSI FADM ASH TLC FAD MOVE FAD FDV MOVEM FMP
(2)	148	8.9	A	FADR MOVEM MOVE FMPR FADR MOVEM ADDI AOJL MOVE FMPR
(3)	222	6.1	A	MOVE FMPR FADR MOVEM
(4)	452	23.7	A	MOVN FMPR FADR MOVN FMPR FADR MOVEM
(5)	181	5.9	A	ADDI AOJL MOVN FMPR FADR MOVN
(6)	226	6.6	A	FMPR FADR MOVEM ADDI AOJL

BASIC Crout

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	2811	36.7	A	FAD	TLZ	CAMGE	POPJ	HRRZ	IMUL	HRRZ
				PUSHJ	MOVE	AOS	MOVE	FAD	TLZ	CAMGE
				POPJ	ADD	ADD	XCT	MOVE	POPJ	
(2)	2811	36.5	A	JSR	JRST	PUSH	LDB	JRST	JRST	MOVSI
				MOVE	HLRZ	PUSHJ	MOVE	AOS	MOVE	FAD
				TLZ	CAMGE	POPJ	HRRZ	IMUL	HRRZ	
(3)	1001	13.7	A	MOVE	POPJ	FMPR	FADR	MOVEM	MOVEI	MOVE
				FADR	JRST	CAMLE	MOVEM	MOVEI	MOVE	MOVEM
				006	JSR	JRST	PUSH	LDB	JRST	
(4)	1239	4.1	A	MOVEI	MOVE	FADR	JRST	CAMLE	MOVEM	
(5)	918	3.5	A	JSR	JRST	PUSH	LDB	JRST	JRST	
(6)	7126	13.7	B	MOVE	FAD	TLZ				
(7)	7126	34.7	B	PUSHJ	MOVE	AOS	MOVE	FAD	TLZ	CAMGE
				POPJ						

Sequences (1) and (2) are largely overlapping parts of the array accessing code. (3) contains most of the general UUO handler in the context of one of the inner product loops, with access to a matrix and some arithmetic. (4) is loop control. Its original time was 5.1% of the total. (5) is the general UUO handler. Its original time was 15.3%. (6) is the abbreviated truncation of indices to integer, (7) shows this in the context of the routine that also checks for index overflow.

BLISS Crout

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence						
(1)	2109	47.9	A	CAMLE	MOVE	IMULI	ADD	ADD	MOVE	IMULI
				ADD	ADD	MOVE	FMPR	FADRB	AOJA	
(2)	361	11.0	A	CAMLE	MOVE	IMULI	ADD	ADD	MOVE	IMULI
				ADD	ADD	MOVE	FMPR	FADRB	AOJA	CAMLE
				JRST	MOVE	SUB	JRST	POP	POP	
(3)	2451	39.8	A	ADD	MOVE	FMPRB	FADRB	AOJA	CAMLE	MOVE
				IMULI	ADD					
(4)	865	4.2	A	PUSH	PUSH	PUSH				
(5)	424	2.8	B	PUSH	PUSH	PUSH	PUSH			
(6)	6010	38.8	B	MOVE	IMULI	ADD	ADD			
(7)	5530	41.1	B	MOVE	IMULI	ADD	ADD	MOVE		
(8)	400	3.0	B	MOVE	IMULI	ADD	ADD	MOVN		

Sequence (1) is the call of ALOG in the beginning of the program, with some environment. (2) is the same as (3) in BLISS Bairstow. (3) is part of the same and reflects bad pruning. (4) to (6) are from the synthetic division and again reflect bad pruning.

5.3.6 Crout

ALGOL Crout

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	1282	19.6	A	AOBJP MOVE MOVE ADDI HLLZ SETZB ROTC EXCH ROTC ROT ANDI HLRZ HRRZ ANDI LSH ANDI LSH CAIN JRST HLRZ
(2)	1001	21.3	A	ADD FMPR MOVE JSP MOVEI JRST MOVEI PUSHJ UFA FADI UFA FADI POPJ MOVEM MOVEM JRST AOS CAMLE MOVE ADD
(3)	819	14.3	A	MOVEM JRST AOS CAMLE MOVE ADD MOVE ADD MOVE MOVE ADD MOVE ADD FMPR MOVE JSP MOVEI JRST MOVEI PUSHJ
(4)	1585	3.6	B	JRST AOS CAMLE
(5)	7351	12.0	A	MOVE ADD
(6)	1225	6.2	A	MOVE ADD FMPR
(7)	3532	11.5	A	MOVE ADD MOVE ADD
(8)	1646	6.6	A	MOVE ADD MOVE ADD MOVE
(9)	1015	6.8	A	MOVE ADD MOVE ADD FMPR

The run time system shows up prominently, as in sequence (1) and others. The double precision add or conversion is (2), part of an innerproduct loop with a call to a double precision routine is shown in (3). (4) is loop control, (5) to (9) are various representations of the matrix and vector access code: (5) is the basic vector access, (7) the basic matrix access, using Iliffe vectors. (6), (8) and (9) are common contexts for these accesses.

(9)	3460	6.3	B	AOJA CAMLE

Sequence (1) shows the inner product loop (two matrixes). (2) shows the same loop with its exit, and exit from the routine. (3) is unknown, maybe part of both inner product loops. (4) and (5) show parts of routine entry, (6) to (8) are forms of the matrix access, (9) is loop control.

FORFOR Crout

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	1225	24.2	A	JFCL FMPR JFCL UFA JFCL FMPI JFCL UFA FADI POP POP POPJ MOVEM MOVEM MOVEI PUSHJ PUSH PUSH UFA FADI
(2)	1015	15.3	A	MOVE MOVEI MOVEM MOVE IMUL MOVE IMUL ADD MOVE ADD MOVN ADD MOVE MOVEI MOVEI MOVEM MOVEM MOVEM PUSHJ PUSH
(3)	2466	18.7	B	MOVE IMUL MOVE IMUL ADD MOVE ADD

The double precision arithmetic is shown in (1), the inner product loop in (2). (3) is access to a formal matrix.

FORTEN Crout

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	819	29.4	A	ADDI AOJL MOVE IMUL ADD ADD MOVE IMUL ADD ADD MOVE FMPR MOVEI MOVEI PUSHJ PUSH PUSH PUSH UFA FADI
(2)	511	3.3	A	MOVE MOVE MOVE MOVE MOVE MOVE
(3)	256	1.7	B	MOVEM MOVEM MOVEM MOVEM MOVEM MOVEM
(4)	735	4.9	B	MOVEM MOVE MOVEM MOVE MOVEM MOVE
(5)	2390	21.2	B	MOVE IMUL ADD ADD MOVE
(6)	2796	21.8	B	MOVE IMUL ADD ADD
(7)	1345	2.1	B	ADDI AOJL

(1) is an innerproduct loop with loop control, access to two matrixes and entry to the double precision routine. (2) to (4) indicate the need for a vider variety of moves. (2) and (3) are from routine entry and exit sequences. (5) and (6) are matrix access. (7) is loop control.

5.3.7 Treesort

This algorithm was chosen because it contains packed data and linked structures. It is the shortest of our subject programs, and the WHILE loop dominates all the results. The only intersting feature is the different way the five systems use to pack information into words. In each case we tried to write the program in a way that the system in question was known to handle efficiently. In the case of FORFOR, therefore, we used division by an octal constant that is a power of 2 to unpack, since this was known to generate a shift. Similarly in the BLISS version we used the bytepointer construct, which generates halfword instructions.

The BASIC result is not compatible with the others for two reasons: A shorter vector was sorted, to reduce execution time, and the vector fetch is very different from in the other systems, as stated elsewhere.

The results were:

ALGOL Treesort:

(1)	8574	18.23	B	MOVE	IDIVI

BASIC Treesort:

(2)	2514	6.5	B	FDVR

BLISS Treesort:

(3)	8174	7.5	B	HLRZ

FORFOR Treesort:

(4)	8974	16.0	B	MOVE	LSH

FORTEN Treesort:

(5)	8174	45.0	B	MOVE	IDIV

5.3.8 PERT

ALGOL PERT

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	555	20.0	A	XCT PUSHJ PUSHJ MOVE PUSH MOVEI MOVE PUSH HLRZ PUSHJ MOVE ADD MOVE POPJ POP POP TLNE POPJ MOVE POPJ
(2)	411	13.6	A	MOVE POPJ POP POP TLNE POPJ MOVE POPJ MOVE MOVE ADD CAME JRST JRST SOS CAIGE XCT PUSHJ PUSHJ MOVE
(3)	487	6.7	B	JRST AOS CAMLE MOVE ADD MOVE ADD MOVE CAIG
(4)	1461	9.5	B	MOVE ADD MOVE ADD
(5)	3415	16.3	B	MOVE ADD MOVE
(6)	622	2.8	B	JRST AOS CAMLE

Sequence (1) is the complete thunk for the parameter to SCAN, including its call by XCT in SCAN, its excursions into the run time support routines, and its return to SCAN. (2) is the loop in SCAN, when the test in the enclosed conditional is false. It overlaps the thunk in (1), but not completely. (3) is the beginning of the loop enclosing the first case statement (switch usage), including loop control. (4) is access code for two level indexing, (5) is the access code for one level indexing in vectors. (6) is loop control.

BASIC PERT

Maximal sequence length: 20.

Seq.	Count	% Time	B/A	Sequence
(1)	874	10.5	A	JRST CAMLE MOVEM MOVEI 005 JSR JRST PUSH LDB JRST JRST MOVSI MOVEI MOVE HRRZ TRNN JRST PUSHJ MOVE AOS
(2)	3989	44.8	A	MOVEI MOVE HRRZ TRNN JRST PUSHJ MOVE AOS MOVE FAD TLZ CAMGE POPJ ADD ADD XCT MOVE POPJ
(3)	874	8.6	A	MOVEI MOVE HRRZ TRNN JRST PUSHJ MOVE AOS MOVE FAD TLZ CAMGE POPJ ADD ADD XCT
(4)	3989	35.7	A	PUSH LDB JRST JRST MOVSI MOVEI MOVE HRRZ TRNN JRST PUSHJ MOVE AOS MOVE

(5)	3115	17.2	A	005 MOVSI	JSR	JRST	PUSH	LDB	JRST	JRST
(6)	1002	4.6	A	JSR	JRST	PUSH	LDB	JRST	JRST	
(7)	926	3.3	B	MOVE	FADR	JRST	CAMLE	MOVEM		

(1) is probably the SCAN loop, showing loop control and entry into the vector fetch UUO. (2) is the body of the vector fetch UUO. (3) overlaps (2) and represents the vector store operations. (4) and (5) are included as examples of bad pruning. (4) overlaps the general UUO mechanism but does not complete the vector fetch sequence of which it is a part. The same holds for (5), which contains the complete UUO mechanism but continues into the fetch. (6) is the UUO mechanism as it should be with good pruning. Its original count was 4991, with 22.9% of the time consumed by it. (7) is loop control.

BLISS PERT

Maximal sequence length: 13 (12).

Seq.	Count	% Time	B/A	Sequence						
(1)	437	12.1	A	ADD MOVE	MOVE	CAME	JRST	SOJG	MOVE	MOVE
(2)	487	12.8	A	AOJA	CAMLE	MOVE	ADD	MOVE	ADD	SKIPG
(3)	399	6.2	A	MOVE	ADD	MOVE	ADD			
(4)	527	8.2	B	MOVE	ADD	MOVE	CAME			
(5)	202	3.1	B	MOVE	ADD	MOVE	MOVEM			
(6)	1716	19.6	B	MOVE	ADD	MOVE				
(7)	996	6.8	B	AOJA	CAMLE					

(1) is the loop in SCAN, when the test is not equal. (2) is the loop control and test of the loop enclosing the first CASE statement. (3) is addition of vector element, or two level indexing. It consumed 14.5% of the time before reduction. (4) to (6) show further variants of vector access, with one or two level indexing. (7) is loop control.

FORFOR PERT

Maximal sequence length: 14 (12).

Seq.	Count	% Time	B/A	Sequence						
(1)	411	14.6	A	ADD	CAME	JRST	CAMGE	AOJA	MOVEM	MOVEI
				ADD	SUB	MOVEM	MOVE	MOVE		
(2)	227	6.8	A	MOVE	JUMPLE	MOVE	CAMGE	AOJA	MOVEM	MOVE
				MOVE	ADD	ADD				
(3)	536	12.5	A	ADD	ADD	MOVEI	HRRM	JSA	MOVM	JRA
(4)	625	10.3	B	MOVEI	HRRM	JSA	MOVM	JRA		
(5)	545	8.8	A	MOVEM	MOVE	MOVE	ADD	ADD		
(6)	1170	15.1	B	MOVE	MOVE	ADD	ADD			
(7)	1725	16.4	B	MOVE	MOVE	ADD				
(8)	481	7.2	A	MOVE	CAMGE	AOJA	MOVEM	MOVE		
(9)	1228	10.9	B	CAMGE	AOJA	MOVEM				

(1) is the loop in SCAN, (2) is the beginning of the loop surrounding the first case (computed GO TO). (3) shows a rather inefficient way of obtaining absolute values, it is shown in its full glory as (4). (5) indicates that vector access with two level indexing may be of importance, this is verified by (6) and (7). (8) shows loop control in context, (9) on its own.

FORTEN PERT

Maximal sequence length: 13 (12).

Seq.	Count	% Time	B/A	Sequence						
(1)	268	11.8	A	ADD	SKIPG	SKIPLE	CAILE	JRST	MOVE	ADD
				MOVE	ADD	MOVE	ADD	MOVM		
(2)	227	6.1	A	ADD	SKIPG	JRST	ADDI	AOJL	MOVE	ADD
				MOVE						
(3)	487	12.1	A	ADDI	AOJL	MOVE	ADD	MOVE	ADD	SKIPG
(4)	411	16.3	A	MOVE	CAME	JRST	AOS	AOSGE	JRST	MOVEI
				ADD	SUB	MOVEM	ADD			
(5)	477	7.4	A	MOVE	ADD	MOVE	ADD			
(6)	1986	22.6	B	MOVE	ADD	MOVE				
(7)	268	4.0	A	MOVE	MOVEM	MOVE	MOVEM			
(8)	913	4.9	B	ADDI	AOJL					

(1) is the body of the CASE statement (computed GO TO), including the preceeding test and the computation of absolute value. (2) is the loop enclosing (1), as seen when the initial test is false. (3) is the same when the test is true and calculation is to proceed as in (1). (4) is

the loop in SCAN. (5) and (6) show the vector accessing code, (7) indicates the need for memory to memory move, (8) is loop control.

5.3.9 Håvie

All the results from this algorithm are dominated by the loop which calls on the integrand, and by the computation of the integrand. The only interesting feature is the use of unrounded an other unusual arithmetic in the mathematical library routines computing SQRT and EXP. We give a few examples of this.

ALGOL Håvie:

Normal arithmetic used.

BASIC Håvie:

(1)	1024	11.6	B	FAD	MOVE	FDV	FAD	FSC
(2)	1024	9.9	B	FDV	FADR	XCT	FSC	

BLISS Håvie:

(3)	512	13.4	B	FSC	MOVEM	FMP	FAD	MOVE
(4)	512	21.2	B	FDV	FAD	FSC	FDV	FADR
(5)	512	10.5	B	FSC	JRST	POP	POP	POP

These are believed to be conesecutive sequences during execution.

FORFOR Håvie:

(6)	1024	21.5	B	FAD	MOVE	FDV	FAD	FSC
(7)	1024	20.8	B	FDV	FADR	FSC	SKIPA	JRA

These are believed to be consecutive. The BLISS mathematical routines were "borrowed" from the FORTRAN library, this explains the similarity of results for these two languages.

FORTEN Håvie:

(8)	1024	17.7	B	MOVE	FDV	FAD	FSC	MOVE
(9)	1024	22.3	B	MOVE	FDV	FADR	FSC	POPJ

5.3.10 Ising

ALGOL Ising

Maximal sequence length: 17.

Seq.	Count	% Time	B/A	Sequence
(1)	983	18.9	A	AOBJP MOVE MOVE ADDI HLLZ SETZB ROTC EXCH ROTC ROT ANDI HLRZ HRRZ ANDI LSH ANDI LSH
(2)	438	7.8	A	LSH JUMPN AND JFFO SKIPN PUSH HRRZ ADDI MOVE CAIG MOVEI SUB HRLI MOVN HRLZI HRRI ADDI
(3)	438	8.4	A	SOJL PUSH HLRZ ANDI LSH HRLZ HLRZ ANDI LSH JUMPN AND JFFO SKIPN PUSH HRRZ ADDI MOVE
(4)	414	8.3	A	MOVE HRRZ ADDM HRRZ XCT CAIE PUSH PUSH HLLZ PUSH MOVEI EXCH HLRZ SOJL PUSH AOJA SOJL
(5)	381	7.2	A	EXCH HLRZ SOJL PUSH AOJA SOJL PUSH AOJA SOJL PUSH HLRZ ANDI LSH HRLZ HLRZ ANDI LSH
(6)	360	7.1	A	HRRZ TLNE JUMPN MOVE MOVE MOVEM MOVEM AOJA AOBJP MOVE MOVE ADDI HLLZ SETZB ROTC EXCH ROTC
(7)	396	5.5	A	CAIN HLRZ ANDI ADD ADD HRRZ TLNE JUMPN MOVE MOVE MOVEM MOVEM AOJA AOBJP
(8)	381	6.6	A	PUSH PUSH HLLZ PUSH MOVEI EXCH HLRZ SOJL PUSH AOJA SOJL PUSH AOJA SOJL
(9)	1044	9.3	A	CAMLE MOVE MOVE ADD MOVE MOVEM JRST AOS
(10)	574	5.1	A	JRST AOS MOVE CAMLE MOVE MOVE ADD MOVE

Sequences (1) through (8) all represent parts of the run time support routines, particularly those used at routine calls and name parameter access. These functions probably account for around 50% of the execution time. (9) and (10) represent parts of some some program loop or loops, possibly the assignment to nonlocal vectors in SORT.

BLISS Ising

Maximal sequence length: 14 (13).

Seq.	Count	% Time	B/A	Sequence
(1)	184	8.4	A	AOJA CAMLE JRST MOVE ADD MOVE ADD AOJ MOVEM AOS MOVE CAMG JRST
(2)	784	19.3	A	CAMLE MOVE MOVE ADD MOVE MOVEM AOJA
(3)	296	6.9	A	MOVE MOVEM CAMLE MOVE MOVE ADD
(4)	381	13.6	A	PUSHJ PUSH PUSH PUSH HRRZ SUBI PUSH
(5)	378	5.7	B	POP POPJ SUB
(6)	281	5.4	B	SUB POP POPJ SUB
(7)	1163	8.0	B	AOJA CAMLE
(8)	1999	15.6	B	MOVE ADD

Here (1) is a piece of the SORT routine, containing the end of one loop, an assignment statement involving a formal vector, and a test ending an outer loop. (2) is from the loops that initialize formal vectors. (3) is probably the initialization of one of these loops and some of the loop. The function entry and exit sequences are represented by (4) through (6), loop control by (7) and formal vector access by (8).

FORFOR Ising

Maximal sequence length: 14.

Seq.	Count	% Time	B/A	Sequence
(1)	112	7.2	A	SUB MOVEM MOVNI ADD MOVE ADD ADD MOVEM MOVE MOVEM MOVE MOVEM CAMGE
(2)	184	10.6	A	MOVEM CAMGE MOVEI ADD MOVE ADD ADD MOVEM AOS MOVE CAMG JRST
(3)	860	15.9	A	MOVE MOVEM CAMGE AOJA
(4)	245	10.3	A	JSA MOVEM MOVEM MOVEI PUSH PUSH PUSH
(5)	248	5.3	A	JRST MOVE MOVE HRROI JRA
(6)	414	6.5	B	JSA MOVEM MOVEM
(7)	657	6.6	B	MOVE ADD

The sequence (1) was not identified. (2) is the same loop as (1) for BLISS Ising, (3) is the vector initialize loops, the vectore in the FORTRAN version being held in COMMON. (4) to (6) represent the calling and exit sequences, (7) gives an idea of the cost of formal vector access.

FORTEN Ising

Maximal sequence length: 16 (15).

Seq.	Count	% Time	B/A	Sequence
(1)	184	13.4	A	MOVE ADD MOVNI ADD ADD MOVEM MOVNI ADD SUB MOVE MOVE MOVEM ADDI AOJL MOVE
(2)	184	9.2	A	MOVE ADD MOVEI ADD ADD MOVEM AOS CAMG JRST MOVE
(3)	860	15.2	A	MOVE MOVEM ADDI AOJL
(4)	360	6.0	A	MOVEI MOVEM MOVEI MOVEM
(5)	657	7.0	B	MOVE ADD
(6)	381	4.4	B	MOVE POPJ
(7)	414	6.1	B	MOVEI PUSHJ MOVEM
(8)	381	5.5	B	JRST MOVE POPJ
(9)	1144	8.4	B	ADDI AOJL

(1) is unknown, but probably in SORT. (2) is the same sequence as (1) in BLISS Ising, (3) is the initialization of the COMMON vectors in SORT, (4) is unknown, (5) is at least in part formal vector access, (6) to (8) is routine entry and exit, and (9) is loop control.

5.4 Sequences applied to data types

Sequences (1) to (6) of the BASIC compiler consume about 30% of the total time of compilation. Much of this could be saved by recoding (1), as previously described. An even larger gain in time would be achieved, however, if the PDP-10 had an instruction to move text (byte strings), with the action to be taken on each byte defined by a table. By a suitable set of options defined by each table entry, this instruction could replace all of the constructs pointed to by sequences (1) to (6). Such an instruction would also reduce space cost compared to the recoded form of (1), and programming cost in any case.

Character handling also shows up in the results from ALGOL, sequences (1) to (8), where it may be assumed to consume well above 10% of the time, and in FORTEN, sequence (6), where it consumes at least 5.5% of the time. We know that all compilers have to perform this kind of processing, the reason it does not show up in the others may be that it is more distributed over the program, and that text lines are not processed as an entity. If an instruction as indicated were provided, compilers would be written to make use of it at a benefit. It can further be safely assumed that it would find application in I/O routines, the importance of such routines is vindicated by our introductory experiments as related on page 37. The need for this type of instruction was also pointed out by Alexander [AleW72].

Another observation we can make from this material is that vector operations are important in many different contexts, and occur to a significant degree in many of our programs: Vector moves consume 4% to 14% of the time in Aitken, 6% to 20% in Ising. Searches in ordered vectors consume 3% to 40% in Aitken, innerproduct consumes 20% to 60% of the time in Crout. Access to vector elements consumes from 5% to 50% in many programs, most in the BASIC programs where they are done through run time system routines.

Hence instructions for vector operators could be introduced to advantage. The least that can be done is to make the vector move operation already existing in the hardware easily available in higher level languages. This is only a first step, however. We propose a vector type along the following lines:

> The concept of vectors with a compile time determined address should be unified with that of dynamically located vectors. They should be given a common formal descriptor and representation.
>
> The descriptor should allow for vectors stored in non consecutive but equidistant locations. Zero should be a legal value for this distance. This would facilitate operations on both coloumns and rows of matrixes; vector moves would perform initialization of a vector with a single value, vector addition would compute the sum of a vector, and so on.
>
> Further, the vectors should be easily combineable into matrixes and access to individual elements of vectors and matrixes should be no more difficult than in common implementations in present systems.
>
> The operators could include moves, searches (possibly binary), vector addition, and inner product, the latter accumulated in double precision.

Possibly this vector type could further be unified with the character string type discussed above.

Other data instructions that might be useful are memory to memory move, and conversion between fixed and floating point numbers. Both of these contribute significantly to the execution time in more than one of our programs. The type conversions have in fact been included in the KI10 processor for the DECsystem 10. This saves 4 to 5 instuctions on each use in a general context, 1 or 2 in the restricted context of BASIC matrix access. For some BASIC programs, this could amount to 3% or 4% of the execution time.

Finally we remark that instructions for packing can save considerable time where they exist in the ISP and are made available by the compiler used. The language PASCAL [WirN71] shows how this can be integrated into a rigid type mechanism.

Two objections against some of these instructions are that they do not easily fit into the PDP-10 instruction format, and the difficulty of accessing them from current higher level languages. The latter problem can, in part at least, be solved by giving them the syntactic status of subroutines. This is already commonly done for operations like negate and absolute value.

5.4.1 Summary

In the previous section we proposed several data types and instructions for inclusion in the PDP-10. For each of these, evidence of its usefulness was found in several algoritms and across most languages. The sequences used to perform these operations were different from language to language, but the underlying operations were the same. This convinced us that our results are valid descriptions of the needs of algorithms. For subject set selection it indicates that the intended area of application should be covered reasonably well, but that the choice of language is less important.

5.5 Properties of operands

As mentioned in the introduction to this chapter, data types desirable for inclusion in the ISP are not only such that are expensive to simulate using existing operators. Other data types might be desirable in order to reduce the space cost of data storage, and to some extent the time cost of the operators.

Examples are given by Wortman [WorD72] and Alexander [AleW72]. They have observed the distribution of written constants in source programs, and found that a large fraction of the integer constants can be held in very few bits. (93% and 56% respectively in 4 bits. The discrepancy may be caused by Wortman's use of student programs, whereas Alexander used larger programs). One would expect a similar observation to hold for dynamic occurrences of integers.

If the operands of each instruction are written on the trace, this dynamic distribution can easily be observed. To relate these observations back to specific storage locations and variables, and to find the maximum space needed for each variable, would require an array equal to the whole data area of the subject program, hence this is a relatively expensive analysis. Furthermore several variables might share the same physical storage location, adding further complication. Hence the utility of a hardware subrange type is not easily determined exactly, although a good indication could be found. We do not do this at present.

To do a similar analysis for floating point types is even harder, since there is no way of telling how much of the accuracy provided is really necessary. This must be left to numerical analysts. A weak indication is provided by observing the usage of immediate type floating point instructions.

Non-uniform distribution of values is not a phenomenon restricted to written integer constants. It has been observed, as reported by Hamming [HamR70] and Pinkham [PinR61], that "naturally occurring numbers" do not have uniformly distributed mantissae. Rather, the mantissae seem to be distributed according to the density function:

$$r(x) = 1/(x * \ln(b)) \qquad (1/b \le x \le 1)$$

where b is the base of the number system. For a binary computer with mantissae in [0.5, 1>, this seems to imply that about 58% of the mantissae would be in [0.5, 0.75>. The essential property of this distribution seems to be its invariance to scale transformations.

Tracing methods can be used to obtain more experimental verification of this, and to evaluate methods designed to exploit it. Other observations of operand values could have relevance for:

- Variable length data types
- Representation of control and addressing information
- Rounding procedures in floating arithmetic

5.6 Data types, Conclusions

In this chapter we have presented various methods for detecting unnecessary data types and operators in existing ISPs, and for detecting non existing but desirable ones.

The former methods are based on frequency counts of instructions, and most of them have also been presented by other workers in the field. Our conclusions about these methods were presented in Section 5.1.3. We pointed out that the results are sensitive to changes both in programming language and algorithm, and hence that a subject set should be well distributed over the area of application and over the languages used.

For the latter problem, we presented a heuristic algorithm for detecting significant dynamic sequences of instructions. This algorithm, including the heuristics, is our work. The algorithm is structured so that the heuristics are easily changed, and new heuristics may be easily added. This method is also applicable to control operators and address calculation.

The results were presented in Section 5.4. They are less dependent on language and algorithms than the frequency results, and properties common to the programs are brought out strongly. This led us to propose several types and operators for inclusion in the ISP that we worked on. A subject set for this method need not represent many languages, but should cover most concepts of the intended area of application.

Finally we propose that desirable data types may also be suggested by a study of the operand values from existing data types. No experimental results from this method are presented.

CHAPTER 6

CONTROL OPERATORS

Our major methods for studying control operators are the same as for data operators, i. e. frequency counts in its various disguises, and instruction sequences. The results of the sequence studies are presented in Section 6.1. We give no comments on the frequency results above those given in Section 5.1.3. We also propose some new methods for use in particular situations. These are discussed in Section 6.2.

Frequency counts indicate that control operators, as defined below, account for a large fraction of the total number of instructions executed (33% by our SNIFT, Figure 5-4). Furthermore, control structures are among the most important means of structuring programs. It follows that efficient implementation of control operators contributes to reduced programming cost as well as time and space cost.

Further motivation for studying control structures and operators is found in the difficulties of compiler writing, particularly in code optimization. A great deal of effort at both compile and run time goes into maintaining (setting and restoring) state information. This applies on subroutine and coroutine calls as well as in more local control contexts where several program branches merge. The inability of compilers to cope with this problem is one of the major reasons for generation of inefficient code. An alternative approach to the problem would be to design ISPs such that the amount of state to be maintained is less, or where it can be saved and restored more efficiently.

Control operators are primarily those which may change the contents of the program counter to a value different from the default value (Old value + 1, n+1'th address etc). Since almost all programs are written in higher level languages, it is reasonable to extend this definition to include instructions used for implementing higher level control structures. Such control structures may be grouped as:

Statement level:

- Unconditional jumps
- Conditionals
- Case selection
- Loops

Program level:
- Subroutines
- Coroutines
- Parallel processes (tasks)

On the program level, program context changes and program communication are most important. Communication ranges from parameter and result passing for subroutines to synchronization for processes.

Our methods are not suited to analysis of programs with processes, since such programs, and certainly the most important ones, have to execute at full speed in order to adequately handle the real time situation they are designed for. The slowdown caused by the tracing interpreter would therefore perturb the results.

There may also be more or less control associated with the operators of the language, ie. the programmer may or may not have to supply explicitly the control necessary for, say, matrix operations, depending on the language (FORTRAN vs. APL). If the control is supplied with the operator, the compiler can in general generate more efficient code, since the context is better defined.

The most important classes of con[illegible]ators on the ISP level may now be described as:
- Unconditional jumps
- Simple tests (implying jumps or skips)
- Loop jumps (count, test and jump)
- Subroutine and return jumps
- Stack manipulating instructions
- Execute instructions
- Some monitor calls
- Other instructions in special contexts

6.1 Sequences applied to control

In this section we discuss those sequences from Section 5.3 that are relevant to control operators.

Most noticeable is the cost of the run-time system for ALGOL programs. This consumes 50%

of the execution time for Ising, 20% for Crout. To achieve a reasonable efficiency for ALGOL programs with many routine calls and name parameters, special instructions and descriptor formats should be introduced. This observation is not new; it has influenced several ISP designs, in particular those of the Burroughs B5000 and its descendants and siblings.

A related feature, more common to all the languages, is the cost of subroutine calls. This is most easily spotted in BLISS programs, since the BLISS calling sequences include stack instructions that are never used in other contexts. In the BLISS compiler calling sequences consume at least 25% of the time, in the FORTEN compiler at least 15%. Both of these compilers were written in BLISS†. In the other programs where we have observations, the time consumed varies between 5% and 20% of the total; 5% in FORTEN Crout, 12% in FORTEN Ising, and over 16% in FORFOR Ising.

The functions performed by these sequences are transmission of parameters and result, manipulation of return linkage, and state setting. The latter includes setting up system registers as well as saving and restoring user registers. The exact constructs needed depend heavily on the language. We present one example:

BLISS programs would execute considerably more efficiently if the PUSHJ and POPJ instructions could manipulate the F register††, and remove the parameters from the stack after exit. The address field of the POPJ instruction, presently unused, could be used to hold the number of parameters, so there would be no space cost at the call site, and the change would fit cleanly into the existing structure. This would reduce the instruction count by 4 in each call, more in some cases. For the BLISS compiler 1/8 of the instruction count would be saved this way; this is about half of the instructions executed in calling sequences. If one were able to specify which registers to save on entry and restore on exit, two further instructions could be saved on each call for each such register. There is, however, no room in the instruction word to specify this. This is a problem common to all calling sequences.

A variant of the subroutine call is the UUO. In our material this is used almost only to call the BASIC run time system. Since this includes vector and array accessing, UUOs are frequently used by BASIC programs, and the central UUO handler of BASIC contributes 15% to 23% of the total execution time. This UUO handler, which consists of 6 instructions,

----

† Two reasons for the difference may be that parameters of FORTEN are passed in registers, or that there are fewer small routines.

†† The F register points to the activation record of the most recently entered routine.

processes the return linkage and selects the right run time routine. Parameters and state are processed at the call site and in the individual routines. Hence the cost of UUOs is extremely high compared to using one of the subroutine call instructions. An exception is when only one UUO is used. In that case the central UUO handler reduces to one instruction. The advantage of UUOs over other subroutine calls is that they allow a memory address (subjected to the standard effective address calculation) and an accumulator address to be transmitted to the routine at no extra cost in space or time at the call site. It also permits linkage to subroutines through a table defined at load time and with no name correspondence. This is of small importance, however. From this we conclude that UUOs should be used only in very special circumstances where the extra time cost is justified. UUOs are also discussed in Section 5.1.3.

Another common construct is loop control. This often consumes no more than 2% to 5% of the execution time, but may consume as much as 9% (Aitken-L) or 10% (FORFOR PERT). It appeared in at least 16 programs, consuming at least 2% of the time in each. In spite of the looping instructions provided in the PDP-10, most loop control sequences consist of two or more instructions. This is primarily due to the fact that most loops count upward to a non zero limit, hence loop control needs to address both the limit and the branch target (assuming the counter to be in a register and the increment to be 1). Contributing are the facts that languages often require the test to be performed at the beginning of the loop but the stepping of the counter at its end, and the need to store the loop counter in memory.

Results reported by Knuth [KnuD70], Shaw [ShaM71], and Alexander [AleW72], for FORTRAN, ALGOL and XPL, show that 93% to 95% of all written counting loops have an increment of one. This form of loop could be done more efficiently in the PDP-10 if the AOBJN (Add one to both, jump if negative) were used. This instruction keeps the loop counter in the right half of a register, the left half is initialized to the negative of the desired number of traversals of the loop. Each time the AOBJN is executed, both halves of the register are incremented by one, and the jump is taken if the result (i.e. the left half) is negative.

This instruction is rarely used in our subject set: 709 times in our 1 million instruction SNIFT. The reason is that extra tests must be performed to make sure that the bound and counter will not overflow the halfword allocated to them. This suggests that two registers should be used, one to hold the upper bound and one for the counter. Our results in Chapter 4 show that there are sufficiently many registers to permit this. Downwards count to a nonzero limit can be handled by a similar instruction.

Commonly used sequences for loop control consist of a AOJXX CAMXX pair. Our instruction will execute in less time than the CAMXX, since no memory operand is needed. Hence these instructions would reduce the time cost of loop control by 40% to 50%, or up to 5% of the execution time of some programs. For very short loops, such as initialization of vectors, this saving could be a significant fraction of the time of the loop. The prologue may imply a larger space cost than for most present loop controls. The hardware cost is that of adding the new instruction(s). The instructions integrate reasonably well into the PDP-10 ISP structure, hence the programming cost will probably be reduced.

We finally draw attention to various forms of testing that are prominent in some of our subject programs. This is seen in the ALGOL run time system and in the compilers, and consumes 2% to 11% of the time. The ALGOL run time system also does a great deal of bit manipulation. We can not suggest any improvements on these operations without further knowledge of their semantics.

6.2 Some special problems

In this section we discuss some problems associated with control operators in general, or with special control operators, which are not easily solved using the more general methods.

6.2.1 Control information

An important aspect of control operations is the control information, i.e. that information which is processed by the normal data operators, but whose main raison d'etre is its use for control purposes. This includes loop counters, stack pointers, return addresses and other addresses, parameter descriptors, displays, etc. Ideas for improved control operators might come from studying how such information is processed.

We make the simplifying assumption that we may disregard information stored in primary memory, and consider only register contents. The information in a register is used for control purposes at the control points, i.e. whenever the register is addressed by a control operator. We are interested in the history of *control information* accumulated at *control points*.

The sequences of sections 5.2 and 6.1 tell us something about this, but they have several deficiencies: They are not accumulated at ***control points***, they contain instructions irrelevant to the ***control information***, and they cover a too short span of time.

Another form of history which we already have is the ***register usage classes*** of Section 4.5. These classes are also inadequate for the present purpose, since only the kinds of events in the life of the register are known, their order and number is unknown.

A third form of history is the sequence of instructions that operated on the specific register before the ***control point*** was reached. Such <u>register sequences</u> can be collected by a process somewhat similar to that described in Section 5.2, but in many ways simpler. Its main properties are:

a) Sequences are accumulated separately for each register, and only instructions affecting that register are included.

b) Each sequence is restricted to one R-life of that register. (R-life defined in Section 4.2). This might cause some sequences (particularly those representing the history of a loop counter) to become very long. A Kleene star kind of concept would be useful in such cases, or the sequences may be truncated at the old end.

c) Sequences are tabulated each time the register is used for a control purpose.

d) The collection takes place in one pass. If space is scarce, some kind of pruning might be necessary.

In such histories, the time order of the events is preserved, but only events affecting the particular register is recorded. If parts of the computation have taken place in other registers, this information is lost. We do not believe this to be a serious problem, however. If it is, one may build the expression trees for the information instead of the sequences. Techniques for doing this are constantly used in compilers, though with the opposite goal. In such trees the exact order of operations is lost, and only those aspects of it are preserved which are relevant to the arithmetic value of the result.

We propose ***register sequences*** as the method for study of control information, most likely to give useful results at a reasonable cost. We have, however, not programmed this method, and hence have no experimental results to support this contention.

6.2.2 Test instructions

To perform a test, 3 addresses are needed: two for the values to compare and one for the instruction that is to be executed if the test succeeds. On the other hand, most ISPs have at most 2 addresses in each instruction (memory address and register or 2 memory addresses). Three techniques are commonly in use to solve this problem:

a) An implicit operand, usually 0, is used for the test. This method is adequate when the value tested either does not have to be computed, or is used for other purposes than testing. This can be studied by ***register sequences***, possibly extended beyond the ***control point***.

b) An implicit change (SKIP), usually 1, 2 or 3, is made in the value of PC depending on the result of the test (succeeds or fails; >, = or <). This may require another 1 or 2 jump instructions to follow the skip instruction, but at most one of these is executed, often none. This method is adequate when the false path is exactly one instruction long, and continues into the true path. Sequences may be used to study the relative frequencies of SKIP JUMP and SKIP NO-JUMP pairs. This requires a modification to the sequence program so that these combinations are always printed before they are pruned. Many SKIP NO-JUMP pairs indicate that this construct is used to advantage.

c) A condition code (CC) is used to store the result of the test. This is subsequently tested by an instruction which specifies the conditional new value of PC in its address field and the desired state of CC in its opcode or register address field. If CC is set by the arithmetic instructions, the first instruction of this pair is not always necessary and this scheme may or may not be more economical in space and time costs than the ones previously described. This method is adequate if the value tested is that most recently computed and it is also used for other purposes.

   If the ISP under study does not use CC's, a few lines of code in the program that accumulates IFT's will simulate a CC. The tables that describe the instructions in terms of the program structure distribution must be available. In this way we may estimate how frequently the introduction of condition codes would have simplified the program.

None of the above methods were implemented; some of the other results, however, have some bearing on these problems.

The program structure distribution, presented in Figure 5-4, indicates that the accumulator is most often tested against memory. The compilers form an exception; here the bit tests and the tests against an immediate operand are more used. The importance of testing against memory may in part be due to the use of these instructions in the loop control. Bit testing and testing against immediate operands are second in importance; tests against 0 are least important. However, testing memory against 0 is as important as the analogous test for the accumulators. Taken together, the tests against zero are almost as important as the accumulator versus memory tests. These results refer to instruction count. In computed time, the tests involving memory increase in relative importance.

We conclude that programmers prefer b) to a), and that they rarely need to test values genuinely for zero, at least not recently computed ones. The memory against zero tests are most common in compilers, this may indicate tests of long lasting status indicators, table entries, etc..

6.3 Control operators, Conclusions

This concludes our discussion of control operators. We have presented the results from the sequence method as applied to control structures, and also suggested some other methods for obtaining additional information. The latter methods, however, have not been implemented.

The detailed implementation of control varies more from language to language than does the use of data operators. This is particularly so for languages that use a run time system for their space allocation and parameter transmission. There is also some variation from algorithm to algorithm due to the different degrees to which the algorithms use certain control structures, and in particular those that involve the run time system. Differences are also inherent in the forms of processing that the algorithms do, as is evident from the program structure distributions in Figure 5-4. We also found significant similarities across languages and algorithms. This is clearly seen in the program structure distribution, and even more clearly in the sequences. In the latter case, though the sequences differ in detail, they reflect common underlying control concepts, and can in many cases be unified. This led us to propose a modification of an existing instruction for loop control, and to point out a basic flaw of the routine call instructions. We also pointed out the inefficiency of the UUO concept of the PDP-10.

If the goal is to detect which control structures are common, the subject set need not represent many languages, but it should be well distributed over all control concepts used in the area of application. However, the detailed implementation of these control concepts is highly language dependent, particularly where a run time system is used. Hence a thorough analysis of programs from the particular language should be done if detailed implementation is the goal.

Our results do in fact suggest that the ISP should have separate control operators, possibly microprogrammed, for each commonly used language.

For the same reasons as when we discussed data types, the generality and consistency of our results lead us to believe in our methods. Our remark in the introduction to this chapter about compilers and state maintenance correlates well with our findings about routine calls. Finally we remark that our results agree well with experience, intuition and afterthought.

CHAPTER 7

ADDRESS CALCULATION

By address calculation (in a wide sense) we mean the calculation of an effective address to operands or instructions in physical memory, based on information provided in the instruction word, in memories addressed by the instruction word, and on other information held in the processor state. Within the problem area so outlined, there are 3 subproblems:

a) Address calculation for data structuring and control operations, which is discussed in Section 7.1. Some of our sequence results are relevant to this problem. These are discussed in Section 7.1.1. We also propose some other methods for special problems in Section 7.1.3. Some of these are closely related to those proposed for control operators in Section 6.2.

b) The problem of mapping a large virtual memory into a small real one. This problem has been addressed by many authors, hence we do not discuss it here, but refer the reader to work mentioned in Section 1.4. The basic idea of these methods is to study the stream of effective addresses, and observe how locality in time implies locality in space.

c) Uniting the need for a large name space with a short address field. We propose no method for this problem; it can be studied by methods similar to those used for b).

7.1 Data structuring

The most common tools in address calculation are indexing, indirection, and base registers. We discuss our methods and results for indirection and indexing. The use of base registers is closely tied to problem c) above. Since we present no methods for this problem, we only mention base registers in passing.

Following a terminology proposed by Foster [FosC70] we will mean by <u>nominator</u> a cell containing an (indirect) address, and by nominee the cell thus addressed. Our other terminology is standard.

7.1.1 Sequences applied to addressing

In this section we discuss those of the sequences in Section 5.2 which are relevant to data structuring, and which indicate the need for more specialized address calculating techniques. Our results reveal two related such structures, namely vectors and matrices.

Vector access consumes 5% or more of the time of at least 14 of our programs, much more in two special cases: 53% in BASIC PERT, and 46% in ALGOL PERT which has a vector element as a name parameter. It consumes more than 10% of the time in Aitken-G, Aitken-L, ALGOL Bairstow, BLISS PERT, FORFOR PERT, FORTEN PERT and BLISS Ising, where more conventional accessing methods are used. In many accesses in PERT the index is itself an indexed variable, a fact which contributes to the cost for that algorithm.

Vector access is particularly time consuming when the base address of the vector is not known to the compiler, that is when the vector is passed as a parameter or when dynamic space allocation is used. The problem could be reduced by addressing vector elements indirectly through a *nominator* whose written address is the base of the vector. This would require that the same index register was used for all accesses to the vector. The compilers that we used do not seem willing to accept this restriction.

In Section 5.4 we proposed the introduction of a vector type to handle vector operations as well as access. Alternatively some other solution, such as the introduction of base registers, should be found to reduce the accessing cost.

The other data structure giving rise to significant sequences is matrices. Matrices are used in Crout, SEC, and Aitken-L. The time cost of accessing was 7% of the total computed time in Aitken-L, and 15% to 20% in SEC. The costs for the versions of Crout are not comparable, due to the special use of UUOs in BASIC, and the non-uniform use of double precision arithmetic which consumes much of the time where used. They were: 11.5% for ALGOL Crout, 60% for BASIC Crout, 39% for BLISS Crout and approximately 20% for the FORTRAN versions. The time advantage of using Iliffe vectors is clearly seen in the ALGOL Crout result.

In many algorithms, such as Crout, the matrix elements are accessed in a systematic manner as row or coloumn vectors. Hence this cost could be reduced by introducing the vector type proposed in Section 5.4 or by adequate language constructs. To speed up genuine random access to matrices, a matrix type with special descriptors and operators could be devised. This should be integrated with the vector type. A step in this direction has been taken in the Burroughs B5000 and related computers. A vector is described by a one word descriptor, the vector so described may itself consist of vector descriptors (i.e. it is an Iliffe vector) and so on.

7.1.2 Indexing and indirection

By observing the frequencies of use of indirection and indexing, we may assess the utility of those features. Thinking the utility of indexing to be above doubt, we did not actually count the number of instructions using it. We did, however, count the number of register lives used for indexing, and we also observed what other kinds of operations those lives were subject to. These are the *register usage classes* of Section 4.5. Our observations are reported in Figure 4-17 and Section 4.5.

We did observe the frequency of use of indirection, and also to how many levels indirection was carried, whether the *nominator* was in a register, and whether pre indexing or post indexing or both were used†.

Two level indirection was observed in all the ALGOL programs, and in FORTEN Crout and FORTEN Ising, the level 2 nominators comprising from about 1/10 to 2/3 of the total number of nominators in these cases. Indirection off byte pointers was found in FORFOR, FORFOR Bairstow, FORFOR PERT and FORFOR SEC, probably associated with I/O, and comprising about 2.6% of the total number of indirect accesses.

Post indexing, was found in the ALGOL programs and in the ALGOL, BASIC and FORFOR compilers. In FORFOR 6.7% of the *nominators* were indexed, in ALGOL PERT 63.8%. For the other programs the percentage ranged between 20 and 50. Our other results are displayed in figures 7-1 through 7-3.

- - - - - - -

† By pre indexing we mean indexing used in the instruction word to access the (first) *nominator*. By post indexing we mean indexing in the *nominator* to access the data or the next nominator.

The low number of indirections through registers indicates that indirection could not be replaced by indexing except at the cost of extra LOAD instructions.

The results for the ALGOL programs indicate that two level indexing may be useful in certain circumstances, for instance where the access path is computed and has a relatively long lifetime, or where it depends on more than one index. Indirection to one level is justified by being used in most programs; one instruction execution is saved on each indirection not through a register. The instruction count of FORTEN Crout would increase by over 7% if indirection were removed, and by 3% or more for 14 of the 41 subject programs.

7.1.3 Addressing information

By addressing information we mean computed information used in address calculation, such as indexes or ***nominators***. The analogy with ***control information*** is obvious, and information about them may be collected in the same way, except that ***addressing information*** is collected at addressing points, defined by analogy to ***control points***. The reader is referred to Section 6.2.1, which applies mutatis mutandis to ***addressing information***.

A study of ***addressing information*** might reveal important manipulation of such information, that could lead to new address calculation algorithms in the ISP. Analysis of ***addressing information*** should be correlated well with that of ***control information***, particularly loop counts and case selectors, which from other experience might be expected to play a double role.

It may also be of interest to study the context of indexed data accesses. Indexing may be used in several contexts, and the following can probably be distinguished mechanically:

Record access, with constant offset and computed base.
Array access, with computed offset and constant base.
Array access, with computed base and computed offset.
Immediate operands.

FIGURE 7-1

Fraction of instructions using indirection

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.006	0.030	0	0.024	0.010
Crout	0.017	0.047	0	0.040	0.073
Treesort	0.000	0.031	0	0.000	0.000
PERT	0.025	0.034	0	0.048	0.034
Håvie	0.019	0.036	0	0.060	0.060
Ising	0.018	-	0	0.032	0.053
Secant	-	-	-	0.034	0.022

Algorithm\Programmer	E	B	A	G	L
Aitken	0	0	0	0	0

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.026	0.015	0.000	0.003	0.000

FIGURE 7-2

Fraction of nominators in a register

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.021	0.007	0	0.024	0.005
Crout	0	0.001	0	0	0.000
Treesort	0	0.001	0	0	0.167
PERT	0.002	0.003	0	0.003	0.001
Håvie	0.001	0.003	0	0.001	0.000
Ising	0.001	-	0	0.048	0.001
Secant	-	-	-	0.171	0.000

Algorithm\Programmer	E	B	A	G	L
Aitken	0	0	0	0	0

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.127	0.999	0.059	0.069	0

FIGURE 7-3

Fraction of indirections pre inde[illegible]

Algorithm\language	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Bairstow	0.359	0.985	0	0.953	0.990
Crout	0.854	0.933	0	1.000	1.000
Treesort	0.600	0.999	0	0.500	0.667
PERT	0.719	0.951	0	0.993	0.998
Håvie	0.661	0.435	0	0.534	0.526
Ising	0.690	-	0	0.737	0.875
Secant	-	-	-	0.828	1.000

Algorithm\Programmer	E	B	A	G	L
Aitken	0	0	0	0	0

Source progr.\Compiler	ALGOL	BASIC	BLISS	FORFOR	FORTEN
Treesort	0.615	0.001	0.937	0.008	0.175

7.1.4 Operand and result modes

Related to addresss calculation is the choice of destination for the result of data operations, and of the order of the operand for non-commutative operators (Examples: Add accumulator to memory, result to memory; Subtract accumulator from memory; etc.). These variants of the operators may be expressed as part of the opcode, or by special addressing modes. If such modes exist on the ISP in question, their utility can be assessed by frequency counts. If such modes do not exist, sequences do not suffice to establish the need for them, since information about the identity of operands is needed. The "result to memory" mode is indicated by the occurrence of OPERATE STORE pairs with the same address. If the accumulator contents is used after such a pair, the indication is for a "result to both" mode. The "inverse order of operand" mode is needed if a large number of LOAD OPERATE pairs exist, where both specify the same accumulator, and the OPERATE is noncommutative and addresses a register for its memory operand.

We did not implement detection of such sequences, and hence have no indications for or against the need for "inverse order of operand" instructions in the PDP-10. The frequency counts in the SNIFT indicate that both the "result to memory" and the "result to both" modes are used, particularly for the commutative operators. Thus FADRB represents 14%, and FADRM 21% of all the occurrences of FADRX instructions† in our SNIFT, a FMPRB represents 2.4% of all the FMPRXs. Similarly the immediate mode for floating arithmetic point is justified, with 6.4% of the FADRXs and 5.4% of the FMPRXs.

7.2 Addressing, Conclusions

The most important part of this chapter discussed those results of our sequence method which applied to address calculation. These results indicated a need for improved accessing methods for matrices, and for vectors with a dynamically determined base address, such as vectors passed as parameters.

We further presented some results from our SNIFT, throwing light on the use of different result destinations for arithmetic operators. Due to our restricted subject set, these latter results are considered inconclusive, but they do suggest a need for the "result to memory" and the "result to both" modes on the PDP-10.

There is nothing in these results to contradict our earlier conclusions about the validity of our methods. We refer the reader to the conclusion sections of chapters 5 and 6, which also apply here, but with some less weight on the dependency of operator implementation on language.

Finally we presented some results on the use of indirection. These show that one level of indirection is certainly useful for our subject set, possibly two. Both pre and post indexing was used.

----

† FADR is floating add with rounding, FMPR is floating multiply with rounding. The suffix X indicates the special mode: Both, Memory or Immediate.

CHAPTER 8

CONCLUSION

In this thesis we have developed some methods for evaluation of the architecture of instruction set processors. The methods are based on analyzing traces of program execution, the traces contain information about every instruction executed by the program. The traces are written as the program is executed on an interpreter for the ISP under investigation. A set of programs, the subject set, is used to represent the workload of the ISP.

The main advantages of these methods are:

a) The level of detail to which they permit us to go. In general every instruction executed, as well as any desirable information from the processor state between instructions, is easily recorded on the trace. If desired, parts of the instruction interpretation may be simulated, and information from this traced. In our case we recorded the instruction word, effective address, program counter, indirect chains, byte pointers and final operands.

b) The general applicability of the methods. The subject set can usually be chosen among any programs that can be compiled into the standard relocatable format used on the processor. The methods are not restricted to a single language or set of languages.

c) The ease of programming of the methods. Other methods could conceiveably provide some of the same information, but would imply a considerable analysis of relocatable programs or core images to reconstruct instruction sequences and register usage.

The subject programs have to be brought into a format acceptable to the interpreter. Usually the standard relocatable format is convenient. For an ISP under design it may therefore be difficult or impossible to obtain a representative subject set. However, in these days of microprogramming, it is not improbable that compilers may be written for an ISP before the ISP itself is frozen. For existing ISPs, as in our experimental work, the interpreter may run on its own ISP. In such cases the relocatable form of the subject programs may be used, and no restrictions are posed on the selection of the subject set.

8.1 Overview of the methods

In chapters 4 through 7 we presented various issues of ISP architecture, viz. register structure, data types and operators, control operators and address calculation. In each chapter we presented methods to deal with these issues, together with experimental results obtained using our subject set.

Some of the methods were the same, or analogous, for several ISP problems. We now review the methods in a methodologically systematic manner. They fall in five categories:

Instruction sequences, with the variant register sequences. Sequences are used to assess the need for new data types and data operators, control operators, and addressing modes. Register sequences (i.e. instruction sequences restricted to instructions affecting one register) can be used for studying control and addressing information in more detail and with greater accuracy than is permitted by the general sequences.

Frequency counts of instruction usage. The instruction frequency table can be displayed in different formats, sorted by execution frequency or by time consumed, grouped into distributions/mixes, or output in the form of the FGR function. From these results we can see which operators were not used, and can be omitted. We can also estimate the cost incurred by having to recode some of the instructions if the instruction set is reduced, and we can see which instructions are candidates for improved implementation.

Register life classification. We showed how to detect register lives (R-lives), and how they could be classified according to the use made of the registers during the lives. This information can be used to assess the need for generality of registers.

Simultaneity of register lives. We presented algorithms to detect how many registers are used simultaneously, and to calculate upper bounds for the time cost incurred if the number of registers were to be reduced while preserving the rest of the ISP structure. These calculations may be done for each of a number of classes of registers, as defined above, as well as for the total set of registers.

Miscellaneous methods. We proposed several special methods for special problems. These can be used to investigate indirection, the utility of condition codes and other solutions to the addressing problem for test instructions, distribution of operand values with partword operands in mind, and so on. One may also implement methods for special properties of the ISP, such as byte pointers on the PDP-10.

The methods have different needs for data space and tables of descriptions. They also use different parts of the trace input. These factors, and also the forms of analysis performed, have some implications for the programming of the methods:

The instruction sequence algorithm makes many passes over the trace, and needs a large data space, but only the instruction word is needed from the trace, and no tables of descriptors are needed. Hence this program should be preceeded by a program that condenses the trace. This latter program can also accumulate the IFT and print its various forms. This latter process requires several tables of descriptors but a moderate amount of data space.

The algorithm for simultaneity of register lives has two phases, the former writing a special file for use by the latter. Neither phase uses much data space, but the first needs some table space. These tables are the same as are used for R-life classification. The latter algorithm needs some data space, but not overly much. Hence it may be programmed with the first phase of the simultaneity algorithm.

In this first phase all register usage, including indexing and indirection through registers, must be detected. For this the effective address is needed. Hence the indirection statistics is best accumulated in this program, and also the special sequences for operand and result modes, if space permits.

To accumulate *register sequences* we need information about the addresses, to see which registers are used, so that the instruction can be associated with the proper register(s). Also, some data space is needed to store the sequences. These sequences can furthermore be collected in one pass. Hence this algorithm does not blend as well with the general sequence algorithm as might be believed at first sight. Many of the same routines and structures can be used, but the main control is different. Hence this method is best programmed separately.

The same holds for operand analysis. For this methods the tables of descriptions used for the Gibson or Program Structure distributions are needed. From the trace, we need the instruction word and the operand words.

8.2 Validity of the methods

In Section 1.2.1 we discussed various methods for collecting dynamic data. It is at this point evident that we could not have obtained our major results without using traces. Both the methods for register structure and the sequence method require the exact sequence of instructions executed. The register results also require the indirect chains and bytepointers as well as the effective rather than the written address of most instructions. This amount of detail, and the preservation of sequentiality which is inherent in tracing, could not be obtained by any of the other methods discussed in Section 1.2.1. Jump tracing could not be used, since we could not have recorded indirect chains or effective addresses that way.

Many of the methods are exact. This applies in particular to the instruction frequency results, the register results up to simultaneity, the register classification results, and the miscellaneous small methods. Hence for these methods the validity of the results depend mostly on the selection of the subject set.

The sequence method is particularly inexact, due to its use of heuristic methods, and to the need for manual analysis. However, the results from this method showed very general results, and many of the sequences found represented general concepts not particular to the language or algorithm where they were found. This supports our contention that these results are valid and useful.

The cost of reducing the number of registers is also inexact, being an upper bound. Our intention was to check these results for some of our BLISS programs. In theory and manuals the BLISS compiler permits the programmer to reserve a number of registers, so that they are not used by the object program except where explicitly named in the source program. However, the compiler refused to generate code for such unwholesome conditions, and the verification could not be done.

Our experimental results show good internal consistency. Many of the results are in general trend independent of both the algorithm and the programming language in which it was coded, and the details often show systematic variation with language and with algorithm. Examples are the register results for ALGOL and BASIC programs, and the use of floating point arithmetic in Bairstow, Crout and Håvie. This is a strong support for their validity.

Some of the results also agree well with previous knowledge - the state maintenance problem for compilers as discussed in Chapter 6 is one example, another is the good agreement of our Gibson distribution with those of Gibson and Gonter.

The dependence on language is most important for those languages that use a run time system for significant parts of their control and accessing functions. In the case of ALGOL, both the sequence results and the register lives were clearly influenced by this. BASIC also influenced the results more than did FORTRAN and BLISS. This is because BASIC uses only one type, because no information is kept in registers between statements, and because a run time system is frequently used. Hence languages with such special properties should be represented in the subject set if they are used. Also, register usage in general depends on language.

Our Aitken results show that the variation due to programmer habits can be large. Analysis of the source programs show that the variation is due mostly to the selection of strategies for subproblems, but that application of coding tricks also plays a part. Our sample is too small to show more than this. The variation is mostly in the sequence results, less in register usage. This suggests that register usage is more a function of the language and compiler used than of the programmer or algorithm.

The register results are not particularly dependent on algorithm. This is natural, since higher level languages hide register usage from the programmer. The choice of algorithm has a strong influence on the use of data operators and data structures.

The results from the FORTRAN programs show good correlation between the two compilers. This may indicate that language has more influence on the object program structure than do compilers. The observation may be peculiar to FORTRAN, which is a well understood language.

A deficiency of the methods in general is that to a large extent they depend on the compilers available for the machine analyzed. A particularly bad or unusual implementation of a commonly used language may flavour a whole analysis, and in no case do the results of an analysis reflect usage of ISP features beyond those that can be made available to programs within the state of the art of compiler writing. On the other hand, the results do indicate what is needed to generate good code for existing languages using existing compiler techniques.

Similarly, if an analysis indicates the need for a new operator or other feature in the ISP, it is not sufficient to implement it in the processor. It must also be made available to the users through the languages they use. This may cause compiler-technical and linguistic problems.

When selecting a subject set for a full scale analysis, care should be taken so that the area of applications is well represented. In particular, all important data structuring methods and special operations should be included. The matrix access of Crout, and the unnormalized arithmetic in certain contexts clearly show this; they are significant where they occur. The individual subject programs should be large enough that the problem of dominating loops is reduced to its right proportions. Good representation of languages is important for register analysis, and particularly for details of control structures and access methods for data structures.. It is less important for data operators.

Another problem occurs when analyzing large programs. How can one represent all aspects of the program within a trace of at most about one million instructions? The obvious solution is a slight modification to the tracer, and possibly the operating system, so that the tracer can be "turned on" for maybe 5000 instructions†, then off for a period of time in which the program executes at full speed, and then on again. Each time the tracer is turned on computation in the subject program has progressed significantly, and different sections of it will be traced. We do not, with this method, have any guarantee that the resulting trace represents a cross section of the program, but our hope is better than by tracing a consecutive tape-full.

8.3 Specific results

We now repeat some of the specific results obtained using our subject set on the PDP-10. We believe most of them generalize to similar ISPs.

> Register utilization was low. The average number of live registers was 7 or less for all programs, the number of registers used was 10 or less 90% of the time for all programs, and 8 or less 98% of the time for 29 of the 41 programs. Time here is the instruction count. If the ISP had only 8 registers, the instruction count of the programs would increase by less than 20% for all programs.
>
> The instruction count of calling sequences can be as high as 25% of the total instruction count. This is particularly noteworthy in view of the common assumption that well structured programs will have many subroutines.

- - - - - - -

† It should be long enough that transients caused by the endpoints are insignificant

The utilization of the opcodes was low. Our subject set used only 27the 4 out of 421 different user instructions. One set of 128 instructions would suffice for 98.8% of the computed time, and a slightly different set of 128 instructions would suffice for 98.6% of the executed instructions. We note in passing that an instruction set of 128 instructions is twice the size of that of the CDC 6000 Central Processor ISP, and about the same size as that of the IBM 360.

Much time was consumed by vector operations or in operations that could be subsumed under a general vector type. This is also true for programs that do not use the mathematical concepts of vectors or matrices. A vector type with sufficiently general operators could be used to advantage by most of our programs. Possibly as much as 30% to 40% of the execution time could be saved in some cases.

We also mention the need for character string operations, and the high cost of using UUOs.

The PDP-10 has a very spacious instruction word, hence both a rich instruction set and a large addressing space. Several of the results above indicate a reduction of the functions in a capability, thus freeing instruction word space. Our suggestions for addition of functions do not nearly consume this space. In fact, the additions indicated could probably be done using the instruction word space which already is available. For an ISP where space is scarce, microprogramming could provide one way of using it efficiently for a given class of applications (See our discussion of the Burroughs B1700, page 15).

8.4 Improvements to the methods

Our present programs could be improved in several ways:

The pruning heuristics used for the sequence collection are not adequate, as discussed in Section 5.2.2. We would expect improved heuristics to significantly reduce the amount of insignificant output from this algorithm, with correspondingly simplified manual analysis.

The results of Figure 4-27 show that we would have achieved a lower cost for reduction of the number of registers if we had pronounced the registers to be dead after a dormancy of only 100 or 60 instructions, instead of 200. An even lower number should be used if the cost is high when using 60.

All of our analysis programs are fairly slow. We believe worthwile reductions in the cost of analysis could be achieved by coding critical routines in machine code, and by cleaning up certain inefficiencies causing extra parameter transmissions.

What is most needed, however, is to try the methods out in a large scale analysis using a significantly larger subject set, where the individual programs also are larger. Only when such an analysis has been successfully completed can we claim that our methods have really proved their worth.

8.4.1 New methods

Some new methods could be implemented. These include the operand analysis, register sequences and other methods outlined in previous chapters, but also one more general one:

Each instruction could be mapped into its generalization in the Program Structure classification, and sequences of such general instructions accumulated. This would bring certain control operations out more clearly, as for example SKIP JUMP sequences, since the conditions on the tests would be suppressed. Also, we could hope to obtain information on common expression forms, generalized calling sequences and loop control, etc.

If the results of such analyses show that the number of sequences found in each analysis is low, and that commonality between algorithms is significant, results of such analyses might be combined to represent the whole subject set, in a way analogous to our present SNIFT.

APPENDIX A

Bibliography

AleW72 Alexander, W. G. How a programming language is used. University of Toronto, Computer Research Group, report CSRG-10, Feb 1972.

AndJ71 Anderson, J. P. Programming language directed machine design - problems and prospects. Proc. symp. on programming and machine organization, IEEE Computer Society, 1971.

ArbR66 Arbuckle, R. A. Computer analysis and thruput evaluation. Computers and automation (Jan 1966), pp. 12-15, 19

BarG68 Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L. and Stokes, R. A. The ILLIAC IV computer. IEEE trans. C-17 vol. 8 pp. 746-757 (Aug. 1968). Also in [BelC71].

BelC71 Bell, C. G. and Newell, A. Computer Structures: Readings and examples. Mc Graw-Hill, N.Y., 1971.

CheP69 Cheng, P. S. Trace-driven system modelling. IBMSJ 8, 4 (1969), 280-289.

CofE68 Coffman, E. G and Varian, L. C. Further experimental data on the behavior of programs in a paging environment. CACM 11, 7 (July 1968), 471-474.

ConW70 Connors, W. D., Mercer, V. S. and Sorlini, T. A. S/360 Instruction usage distribution. Report IBM-SDS TR 00.2025, May 8, 1970.

DEC71 DECsystem10 Assembly language handbook, second edition. (DEC-10-NRZBD); Digital Equipment Corp., Maynard, Mass. 01754, 1971.

FosC70 Foster, C. C. Computer architecture. Van Nostrand Reinhold Co., New York, N. Y., 1970.

FosC71a Foster, C. C, Gonter, R. H and Riseman, E. M. Measures of op-code utilisation. IEEE transactions on computers 20, 5 (May 1971), 582-584.

FosC71b Foster, C. C and Gonter, R. Conditional interpretation of operation codes. IEEE transactions on computers 20, 1 (Jan 1971), 108-111.

FosC72 Foster, C. C. and Riseman, E. M. Percolation of code to enhance parallel dispatching and execution. IEEE trans. C-21, pp. 1411-1415, Dec. 1972.

GibD67 Gibson, D. H. Considerations in block oriented systems design. AFIPS Conference proceedings vol. 31 (SJCC 1967), Thompson book co., Washington D. C., 1967, pp. 75-80.

GibJ70 Gibson, J. C. The Gibson mix. Report TR 00.2043, IBM Systems Development Div, Poughkeepsie, N.Y., 1970. Research done in 1959.

GonR69 Gonter, R. H. Comparison of the Gibson mix with UMASS mix. Publication no. TN/RCC/004, University of Massachusetts, Research Computing Center.

HamR70 Hamming, R. On the distribution of numbers. Bell systems tecnical journal 49 (Oct 1970), 1609-1625.

HatD72 Hatfield, D. J. Experiments on page-size, program access patterns and virtual memory performance. IBMJRD 16, 1 (Jan 1972), 58-66.

HerE55 Herbst, E. H, Metropolis, N and Wells, M. B. Analysis of problem codes on the MANIAC. MTOAC 9 (Jan 1955), 14-20.

HolS71 Holland, S. A. and Purcell, C. J. The CDC STAR-100: A large scale network oriented computer system. Proc 1971 IEEE Conference, pp. 55-56.

IBM56 The FORTRAN automatic coding system for the IBM 704 EDPM, (Programmers Reference Manual), IBM Corp. 32-7026, Oct. 1956.

KapK71 Kaplan, K. R. Cache system studies. RCA reprint PE-518., RCA David Sarnoff Research Ctr., 1971.

KemJ61 Kemeny, J. G. and Kurtz, T. E. BASIC Dartmouth College computation center, June 1961.

KnuD69 Knuth, D. E. The Art of Computer Programming Vol. 1: Fundamental Algorithms.. Addison-Wesley, Reading, Mass., 1969.

KnuD70 Knuth, D. E. An empirical study of FORTRAN programs. Report CS-186, Stanford University, Computer Science Dept., Stanford, Calif., 1970. Also AD 715 513

LewP71 Lewis, P. A. W and Yue, P. C. Statistical analysis of program reference patterns in a paging environment. IEEE Computer Conference digest, Sept 1971. Also IBM report.

McKW67 McKeeman, W. M. Language directed computer design. AFIPS FJCC Proc. vol. 31, pp. 413-417, 1967.

McKW70 Mc Keeman, W. M., Horning, J. J. and Wortman, D. B. A compiler generator. Prentice-Hall, Englewood Cliffs, N.J. 1970.

MilW49 Milne, W. E. Numerical calculus. Princeton University Press, 1949.

MurJ70 Murphey, J. O. and Wade, R. M. The IBM 360/195 in a world of mixed job streams. DATAM (Apr. 1970), 72-79.

NauP63 Naur, P. (Ed.) Revised Report on the Algorithmic language ALGOL 60, CACM 6.1 (Jan 1963), pp. 1-17.

PinR61 Pinkham, R. S. On the distribution of first significant digits. Ann. math. stat. vol. 32, pp. 1223-1230, 1961.

RaiE66 Raichelson, E and Collins, G. A method for comparing the internal operating speeds of computers. CACM 7, 5 (May 1966), 309-310.

RisE72 Riseman, E. M. and Foster, C. C. The inhibition of potential parallelism by conditional jumps. IEEE trans C-21, pp. 1405-1411, Dec. 1972.

SaaH72 Saal, H. J. and Shustek, L. J. Microprogrammed implementation of computer measurement techniques. 5th. ann. workshop on microprogramming preprints, ACM and IEEE 1972.

SelL.nd Seligman, L. Experimental data for the working set model. Memo #39, MIT project MAC, Computation Structures Group.

ShaM71 Shaw, M. M. Language structures for contractible compilers. Ph.D. thesis, Carnegie-Mellon University, Computer Science Dept., Pittsburgh, Pa. 15213, Dec 1971.

SteJ.nd Stewart, J. A. Selecting and evaluating a medium scale computer system. Louisiana state university at New Orleans, LUNSO computer research center. Date approximately 1969.

ThoJ64 Thornton, J. E. Parallel operation in the Control Data 6600. AFIPS FJCC Proc. 1964, vol. 26, part 2, pp. 33-40.

TjaG70 Tjaden, G. S. and Flynn, M. J. Detection and parallel execution of independent instructions. IEEE trans. C-19, pp. 889-895, Oct. 1970.

USAS66 USA Standard FORTRAN, United States of America Standards Institute, USAS X3.9-1966, New York, N. Y. Mar 1966.

WicB69 Wichmann, B. A. A comparison of ALGOL-60 execution speeds. CCU report no. 3, National Physical Laboratories, Central Computer Unit, Teddington, Middlesex, England, Jan 1969.

WicB70 Wichmann, B. A. Some statistics from ALGOL programs. CCU report no. 11, National Physical Laboratories, Central Computer Unit, Teddington, Middlesex, England, Aug 1970.

WilW72a Wilner, W. T. Design of the Burroughs B1700. AFIPS FJCC Proc., vol. 41, pp. 489-497, 1972.

WilW72b Wilner, W. T. Burroughs B1700 memory utilization. AFIPS FJCC Proc., vol. 41, pp. 579-586, 1972.

WinR71 Winder, R. O. Data base for computer performance evaluation. RCA-reprint PE-517, RCA David Sarnoff Research Ctr., 1971.

WinR73 Winder, R. O. A data base for computer evaluation. Computer vol. 6 no. 3, pp. 25-29, March 1973.

WirN71 Wirth, N. The Programming language Pascal. Acta Informatica, vol. 1, pp. 35-68, 1971.

WirN72 Wirth, N. On PASCAL, code generation, and the CDC 6000 computer. Stanford University, Computer Science dept., STAN-CS-72-257, Feb. 1972.

WorD72 Wortman, D. B. A study of language directed machine design. Ph.D. thesis, Stanford University, Computer Science dept. 1971. University of Toronto, Computer research group, report CSRG - 20, Dec 1972.

WulW70 Wulf, W. A. et. al. BLISS reference manual. Carnegie-Mellon University, Computer Science Dept., Pittsburgh, Pa. 15213, Jan 1970.

WulW72 Wulf, W. A. and Bell, C. G. C.mmp-A multi-mini processor. AFIPS FJCC Proc. vol. 41, part 2, pp. 765-777, 1972.

APPENDIX B

The register usage classification

```
EACH INSTRUCTION IS DESCRIBED BY A TWO WORD TABLE ENTRY.
THE VALUES OF THESE ENTRIES ARE DEFINED BY COMBINATIONS OF THE FOLLOWING SYMBOLS:

; OFFSETS FOR FIELDS, WORD 1.

ACCOFF==1;               LAST BIT OF ACCUMULATOR USAGE FIELD.
INXOFF==ACCOFF*20;       LAST BIT OF INDEX USAGE FIELD.
MAOFF==INXOFF*10;        LAST BIT OF EFF.ADDR. USAGE FIELD
AAROFF==MAOFF*10;        LAST BIT OF ACC-ARITHMETIC FIELD.
MAROFF==AAROFF*10;       LAST BIT OF M-ARITHMETIC CODE.

; OFFSETS FOR FIELDS, WORD 2.

RCOFF==1;                LAST BIT OF SUB-OP DESCR FOR REGISTER.
MCOFF==10000;            LAST BIT OF SUB-OP DESCR FOR MEMORY.


; REFERENCE ATTRIBUTES

; ACCUMULATOR  FIELD

ACCNUS=0*ACCOFF;         ACCUMULATOR NOT USED
ACCNZL=1*ACCOFF;         ACCUMULATOR RELOADED IF NOT ACC 0.
ACCLOD=2*ACCOFF;         ACCUMULATOR ALWAYS RELOADED
ACCMOD=3*ACCOFF;         ACCUMULATOR MODIFIED.
ACCMM2=4*ACCOFF;         ACC, ACC+1 BOTH MODIFIED
ACCUSE=5*ACCOFF;         ACC VALUE USED, NOT CHANGED.
ACCUU2=6*ACCOFF;         ACC, ACC+1 BOTH USED.
ACCUND=7*ACCOFF;         UNDEFINED (AS IN CALL, CALLI ETC.)
ACCUL2=10*ACCOFF;        ACC USED, ACC+1 LOADED
ACCML2=11*ACCOFF;        ACC MODIFIED, ACC+1 LOADED

; EFFECTIVE MEMORY ADDRESS FIELD

MNUSED=0*MAOFF;          M NOT USED
MUSED=1*MAOFF;           M USED BUT NOT CHANGED
MLOAD=2*MAOFF;           M LOADED W. NEW VALUE
MMODIF=4*MAOFF;          M MODIFIED
MSPECL=5*MAOFF;          M NEEDS SPECIAL TREATMENT.
MFILDA=6*MAOFF;          M USED FOR FILE DESCRIPTOR.
MNZAUS=7*MAOFF;          M USED IF ACC NOT ACC 0.


; ACCESS ATTRIBUTES:

; INDEX REGISTER USAGE

INXNUS=0*INXOFF;         INDEX NOT USED
INXDAT=1*INXOFF;         INDEX USED FOR DATA INDEXING
INXJMP=2*INXOFF;         INDEX USED FOR JUMP INDEXING, INCL. XCT.
INXIMM=4*INXOFF;         INDEX USED FOR IMMEDIATE OPERAND.

; ARITHMETIC WHEN RESULT TO ACCUMULATOR

AARNONE=0*AAROFF;        NO ARITHMETIC
AARCOU=1*AAROFF;         FIXP. ADD/SUB (COUNTER REGISTER)
AARFIX=2*AAROFF;         FIXP. MUL/DIV
AARFLO=4*AAROFF;         FLOATING POINT OPERATIONS.

; ARITHMETIC WHEN RESULT TO MEMORY

MARNON=0*MAROFF;         NO ARITHMETIC RESULT TO MEMORY
MARCOU=1*MAROFF;         COUNTER OPERATIONS (FIXP. +-)
MARFIX=2*MAROFF;         FIXPOINT OPERATIONS
MARFLO=4*MAROFF;         FLOATING OPERATIONS

; NON-ARITHMETIC ACCUMULATOR OPERATIONS

RCNONE=0*RCOFF;          NOT USED
RCSTOR=1*RCOFF;          VALUE IN ACC STORED
RCHWMO=2*RCOFF;          ONE HALFWORD LOADED OTHER UNCHANGED.
RCBYTE=4*RCOFF;          BYTE LOADED/DEPOSITED
RCLOGI=10*RCOFF;         LOGICAL OPERATIONS
RCSHIF=20*RCOFF;         ACC. SHIFTED
RCSTAK=40*RCOFF;         ACC. USED AS STACK POINTER
RCADDR=100*RCOFF;        ACC USED FOR ADDRESS (AS IN BLT).
RCTEST=200*RCOFF;        ACC. TESTED UPON
```

```
; NON ARITHMETIC MEMORY OPERATIONS

MCNONE=0*MCOFF;          MA NOT USED.
MCSTOP=1*MCOFF;          MA IS STORED (PUSHED).
MCHWMD=2*MCOFF;          MA HALFWORD MODIFIED.
MCBYTE=4*MCOFF;          MA BYTE MODIFIED.
MCLOGI=10*MCOFF;         MA MODIFIED BY LOGICAL OPERATION.
MCTEST=200*MCOFF;        MA TESTED
MCMONI=400*MCOFF;        MA USED FOR MONITOR PARAMETER, POSSIBLY MODIFIED
MCBPTR=1000*MCOFF;       MA USED FOR BYTE POINTER
MCINDA=2000*MCOFF;       MA USED FOR INDIRECT ADDRESS.
MCEXEC=4000*MCOFF;       MA EXECUTED (JUMP OR EXEC).
```

SAMPLE INSTRUCTION DESCRIPTIONS

```
FDVB:  (FLOATING DIVIDE, RESULT TO ACCUMULATOR AND MEMORY)
WORD 1: ACCMOD+INXDAT+MMODIF+AARFLO+MARFLO
WORD 2: RCSTOP

MULI:  (MULTIPLY IMMEDIATE):
WORD 1: ACCML2+INXIMM+MNUSED+AARFIX
WORD 2: 0

ADJX:  (ADD ONE TO ACCUMULATOR, JUMP IF X):
WORD 1: ACCMOD+INXJMP+MUSED+AARCOU
WORD 2: MCEXEC+RCTEST
```

APPENDIX C

Output from register classification program

Fixpoint addition and subtraction are referred to as counter operations.

The non obvious encodings of the usage parameters are:

COFIX	Counter and fixpoint arithmetic	XDAIA	Indexing data access
FXFLO	Fixed and floating point	XIMJM	Indexing immediate and jumps
COFLO	Counter and floating point	XDAJM	Indexing data access and jumps
CFXFL	Counter, fixed and floating	XDJIM	Indexing data accesses, jumps and immediate
XDATA	Indexing data accesses		
XJUMP	Indexing jumps		
XIMME	Indexing immediate operands		

MASK refers to the class definition given in the output procedure.

UNION CLASS is the union of all classes listed above it.

THE FULL UCT

MASK AND ITS COMPLEMENT
777777 CFXFL XDJIM STOPE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESTS MONIT BYTPT INDPK EXECT
000000

CLASS	COUNT	FRAC-TION	CUMUL. FRACT.	AVRGE LENGTH	INTERPRETATION													
000104	1349	0.220	0.220	5.55	FLOAT		STOPE											
000004	712	0.116	0.336	3.63	FLOAT													
000000	618	0.101	0.437	1.31														
000010	551	0.090	0.527	3.26		XDATA												
000100	545	0.089	0.616	7.90			STOPE											
020000	487	0.079	0.695	2.46										TESTS				
000130	280	0.046	0.741	9.21		XDAJM	STOPE											
020001	182	0.030	0.770	10.59	COUNT									TESTS				
000012	150	0.024	0.795	2.52	FIXPT	XDATA												
000011	126	0.021	0.815	4.49	COUNT	XDATA												
020111	118	0.019	0.835	53.51	COUNT	XDATA	STOPE							TESTS				
000101	109	0.018	0.852	4.70	COUNT		STOPE											
020500	108	0.018	0.870	12.00			STOPE		BYTOP					TESTS				
000420	84	0.014	0.884	7.00		XJUMP			BYTOP									
021400	60	0.010	0.893	9.60					BYTOP	LOGIC				TESTS				
022101	56	0.009	0.902	6.21	COUNT		STOPE				SHIFT			TESTS				
401000	39	0.006	0.909	4.05						LOGIC								EXECT
020101	37	0.006	0.915	4.00	COUNT		STOPE							TESTS				
000110	36	0.006	0.921	14.44		XDATA	STOPE											
004000	22	0.004	0.924	68.23								STACK						
020104	20	0.003	0.928	26.00	FLOAT		STOPE							TESTS				
021401	18	0.003	0.931	15.00	COUNT				BYTOP	LOGIC				TESTS				
000411	18	0.003	0.933	4.00	COUNT	XDATA			BYTOP									
021000	16	0.003	0.936	7.94						LOGIC				TESTS				
020400	16	0.003	0.939	5.75					BYTOP					TESTS				
020003	15	0.002	0.941	16.60	COFIX									TESTS				
020100	15	0.002	0.944	29.40			STOPE							TESTS				
000020	13	0.002	0.946	2.46		XJUMP												
000500	13	0.002	0.948	20.54			STOPE		BYTOP									
021042	12	0.002	0.950	11.00	FIXPT	XIMME				LOGIC				TESTS				
023104	12	0.002	0.952	29.83	FLOAT		STOPE			LOGIC	SHIFT			TESTS				
000103	12	0.002	0.954	22.50	COFIX		STOPE											
022103	12	0.002	0.956	6.50	COFIX		STOPE				SHIFT			TESTS				
002104	12	0.002	0.958	19.33	FLOAT		STOPE				SHIFT							
000205	12	0.002	0.960	10.00	COFLO			HALFW										
020151	12	0.002	0.962	52.42	COUNT	XIMDA	STOPE							TESTS				
000102	11	0.002	0.963	2.00	FIXPT		STOPE											
020002	9	0.001	0.965	14.11	FIXPT									TESTS				
023100	8	0.001	0.966	22.00			STOPE			LOGIC	SHIFT			TESTS				
020010	8	0.001	0.967	7.25		XDATA								TESTS				
004100	8	0.001	0.969	38.00			STOPE					STACK						
000603	7	0.001	0.970	8.00	COFIX			HALFW	BYTOP									
010000	7	0.001	0.971	2.00									ADDRS					
010300	7	0.001	0.972	7.00			STOPE	HALFW					ADDRS					
000400	7	0.001	0.973	90.00					BYTOP									
021300	7	0.001	0.974	93.00			STOPE	HALFW		LOGIC				TESTS				
020401	7	0.001	0.976	8.00	COUNT				BYTOP					TESTS				
000300	7	0.001	0.977	23.86			STOPE	HALFW										
002610	7	0.001	0.978	18.14		XDATA		HALFW	BYTOP		SHIFT							
021170	7	0.001	0.979	66.29		XDJIM	STOPE			LOGIC				TESTS				
010001	7	0.001	0.980	3.00	COUNT								ADDRS					
000002	7	0.001	0.981	7.43	FIXPT													
022021	6	0.001	0.982	13.00	COUNT	XJUMP					SHIFT			TESTS				
320350	6	0.001	0.983	76.00		XIMDA	STOPE	HALFW						TESTS		BYTPT	INDPK	
021500	6	0.001	0.984	14.00			STOPE		BYTOP	LOGIC				TESTS				
001203	6	0.001	0.985	39.00	COFIX			HALFW		LOGIC								
401200	6	0.001	0.986	5.00				HALFW		LOGIC								EXECT
000112	6	0.001	0.987	52.00	FIXPT	XDATA	STOPE											
220001	6	0.001	0.988	8.00	COUNT									TESTS			INDPK	
022121	6	0.001	0.989	62.00	COUNT	XJUMP	STOPE				SHIFT			TESTS				
023107	6	0.001	0.990	17.00	CFXFL		STOPE			LOGIC	SHIFT			TESTS				
000430	6	0.001	0.991	4.00		XDAJM			BYTOP									
021100	6	0.001	0.992	6.00			STOPE			LOGIC				TESTS				

Output from register classification program

```
004010       6  0.001  0.993  11.17          XDATA                                    STACK
021220       6  0.001  0.994  14.17          XJUMP        HALFW       LOGIC                              TESTS
023106       6  0.001  0.995  14.00  FXFLO         STOPE              LOGIC SHIFT                        TESTS
023102       5  0.001  0.996  29.00  FIXPT         STORE              LOGIC SHIFT                        TESTS
000001       3  0.000  0.996  10.67  COUNT
004001       2  0.000  0.997   3.00  COUNT                                            STACK
022011       2  0.000  0.997  83.50  COUNT XDATA                            SHIFT                        TESTS
000111       2  0.000  0.997  14.50  COUNT XDATA STORE
000030       2  0.000  0.998   6.00        XDAJM
020013       2  0.000  0.99P   6.00  COF1X XDATA                                                         TESTS
002600       1  0.000  0.998  31.00                   HALFW BYTOP       SHIFT
000070       1  0.000  0.990  45.00        XOJIM
000200       1  0.000  0.998  28.00                   HALFW
004110       1  0.000  0.999  44.00        XDATA STOPE                                STACK
000703       1  0.000  0.999  13.00  COF1X       STOPE HALFW BYTOP
002101       1  0.000  0.999   5.00  COUNT       STOPE                  SHIFT
000050       1  0.000  0.999   6.00        XIMDA
020501       1  0.000  0.999  57.00  COUNT       STOPE       BYTOP                                       TESTS
000151       1  0.000  0.999  16.00  COUNT XIMDA STOPE
020011       1  0.000  1.000  42.00  COUNT XDATA                                                         TESTS
010200       1  0.000  1.000   9.00                   HALFW                                 ADDPS
400000       1  0.000  1.000  19.00                                                                                       EXECT
400600       1  0.000  1.000  12.00                   HALFW BYTOP                                                         EXECT

                 6133 LIFETIMES.   86 DIFFEPENT CLASSES.

UNION CIASS AND ITS COMPLEMENT
737777                               CFXFL XOJIM STOPE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESIS       BYTPT INOPK EXECT
040000                                                                                           MONIT
```

CLASSES USED FOR INDEXING

```
MASK AND ITS COMPLEMENT
000?.0                                     XDJIM
77?707                               CFXFL       STOPE HALFW BYTOP LOGIC SHIFT STACK ADDPS TESTS MONIT BYTPT INOPK EXECT

                FPAC- CUMUL.  AVPGE
 CLASS   COUNT   TION FPACT. LENGTH  INTEPPPETATION

000010     551  0.090  0.090   3.26        XDATA
000130     280  0.046  0.135   9.21        XDAJM STORE
000012     150  0.024  0.160   2.52  FIXPT XDATA
000011     126  0.021  0.100   4.49  COUNT XDATA
020111     118  0.019  0.200  53.51  COUNT XDATA STOPE                                                   TESTS
000420      84  0.014  0.213   2.00        XJUMP             BYTOP
000110      36  0.006  0.219  14.44        XDATA STOPE
000411      18  0.003  0.222   4.00  COUNT XDATA             BYTOP
000020      13  0.002  0.224   2.46        XJUMP
021042      12  0.002  0.226  11.00  FIXPT XIMME                   LOGIC                                 TESTS
020151      12  0.002  0.228  52.42  COUNT XIMDA STOPE                                                   TESTS
020010       8  0.001  0.230   7.25        XDATA                                                         TESTS
002610       7  0.001  0.231  18.14        XDATA       HALFW BYTOP       SHIFT
021170       7  0.001  0.232  66.29        XDJIM STOPE             LOGIC                                 TESTS
022021       6  0.001  0.233  13.00  COUNT XJUMP                         SHIFT                           TESTS
320350       6  0.001  0.234  76.00        XIMDA STOPE HALFW                                             TESTS       BYTPT INDPK
000112       6  0.001  0.235  52.00  FIXPT XDATA STOPE
022121       6  0.001  0.236  62.00  COUNT XJUMP STOPE                   SHIFT                           TESTS
000430       6  0.001  0.237   4.00        XDAJM             BYTOP
004010       6  0.001  0.238  11.17        XDATA                               STACK
021220       6  0.001  0.239  14.17        XJUMP       HALFW       LOGIC                                 TESTS
022011       2  0.300  0.239  83.50  COUNT XDATA                         SHIFT                           TESTS
000111       2  0.000  0.239  14.50  COUNT XDATA STOPE
000030       2  0.000  0.240   6.00        XDAJM
020013       2  0.000  0.240   6.00  CDF1X XDATA                                                         TESTS
000070       1  0.000  0.240  45.00        XDJIM
004110       1  0.000  0.240  44.00        XDATA STOPE                         STACK
000050       1  0.000  0.241   6.00        XIMDA
000151       1  0.000  0.241  16.00  COUNT XIMDA STORE
020011       1  0.000  0.241  42.00  COUNT XDATA                                                         TESTS

                 1477 LIFETIMES.   30 DI FEPENT CLASSES.

UNION CLASS AND ITS COMPLEMENT
327773                               COF   XOJIM STOPE HALFW BYTOP LOGIC SHIFT STACK       TESTS       BYTPT INOPK
450004                               FLOA                                            ADDPS       MONIT             EXECT
```

Output from register classification program

THE ARITHMETIC CLASSES

CLASS: NO ARITHMETIC ** *

MASK AND ITS COMPLEMENT
000007 CFXFL
777770 XDJIM STORE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESTS MONIT BYTPT INDRK EXECT

CLASS	COUNT	FRAC- TION	CUMUL. FRACT.	AVRGE LENGTH	INTERPRETATION
000000	618	0.101	0.101	1.31	
000010	551	0.090	0.191	3.26	XDATA
000100	545	0.089	0.279	7.98	STORE
020000	487	0.079	0.359	2.46	TESTS
000130	280	0.046	0.405	9.21	XDAJM STORE
020500	108	0.018	0.422	12.00	STORE BYTOP TESTS
000420	84	0.014	0.436	2.00	XJUMP BYTOP
021400	60	0.010	0.446	9.60	BYTOP LOGIC TESTS
401000	39	0.006	0.452	4.05	LOGIC EXECT
000110	36	0.006	0.458	14.44	XDATA STORE
004000	22	0.004	0.461	68.23	STACK
021000	16	0.003	0.464	7.94	LOGIC TESTS
020400	16	0.003	0.467	5.75	BYTOP TESTS
020100	15	0.002	0.469	29.40	STORE TESTS
000020	13	0.002	0.471	2.46	XJUMP
000500	13	0.002	0.473	20.54	STORE BYTOP
023100	8	0.001	0.475	27.00	STORE LOGIC SHIFT TESTS
020010	8	0.001	0.476	7.25	XDATA TESTS
004100	8	0.001	0.477	30.00	STORE STACK
010000	7	0.001	0.478	2.00	ADDRS
010300	7	0.001	0.480	7.00	STORE HALFW ADDRS
000400	7	0.001	0.481	90.00	BYTOP
021300	7	0.001	0.482	93.00	STORE HALFW LOGIC TESTS
000300	7	0.001	0.483	23.86	STORE HALFW
002610	7	0.001	0.484	18.14	XDATA HALFW BYTOP SHIFT
021170	7	0.001	0.485	66.29	XDJIM ST[illegible] LOGIC TESTS
320350	6	0.001	0.486	76.00	XIMDA STORE HALFW TESTS BYTPT INDRK
021500	6	0.001	0.487	14.00	STORE BYTOP LOGIC TESTS
401200	6	0.001	0.488	5.00	HALFW LOGIC EXECT
000430	6	0.001	0.489	4.00	XDAJM BYTOP
021100	6	0.001	0.490	6.00	STORE LOGIC TESTS
004010	6	0.001	0.491	11.17	XDATA STACK
021220	6	0.001	0.492	14.17	XJUMP HALFW LOGIC TESTS
000030	2	0.000	0.492	6.00	XDAJM
002600	1	0.000	0.493	31.00	HALFW BYTOP SHIFT
000070	1	0.000	0.493	45.00	XDJIM
000200	1	0.000	0.493	28.00	HALFW
004110	1	0.000	0.493	44.00	XDATA STORE STACK
000050	1	0.000	0.493	6.00	XIMDA
010200	1	0.000	0.493	9.00	HALFW ADDRS
400000	1	0.000	0.494	19.00	EXECT
400600	1	0.000	0.494	12.00	HALFW BYTOP EXECT

3028 LIFETIMES. 42 DIFFERENT CLASSES.

UNION CLASS AND ITS COMPLEMENT
737770 XDJIM STORE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESTS BYTPT INDRK EXECT
040007 CFXFL MONIT

```
CLASS:   FIXPOINT ADD AND SUBTRACT   • • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •

MASK AND ITS COMPLEMENT
000001                                     COUNT
777776                                     FXFLO XDJIM STORE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESTS MONIT BYTPT INOPK EXECT

                      FRAC- CUMUL.  AVRGE
 CLASS    COUNT        TION FRACT. LENGTH  INTERPRETATION

020001      182  0.030  0.030  10.59  COUNT                                                 TESTS
000011      126  0.021  0.050   4.49  COUNT XDATA
020111      118  0.019  0.069  53.51  COUNT XDATA STORE                                     TESTS
000101      109  0.018  0.087   4.70  COUNT       STORE
022101       56  0.009  0.096   6.21  COUNT       STORE                   SHIFT             TESTS
020101       37  0.006  0.102   4.00  COUNT       STORE                                     TESTS
021401       18  0.003  0.105  15.00  COUNT                   BYTOP LOGIC                   TESTS
000411       18  0.003  0.108   4.00  COUNT XDATA             BYTOP
020003       15  0.002  0.111  16.60  COFIX                                                 TESTS
000103       12  0.002  0.113  22.50  COFIX       STORE
022103       12  0.002  0.115   6.50  COFIX       STORE                   SHIFT             TESTS
000205       12  0.002  0.117  10.00  COFLO             HALFW
020151       12  0.002  0.119  52.42  COUNT XIMDA STORE                                     TESTS
000603        7  0.001  0.120   8.00  COFIX             HALFW BYTOP
020401        7  0.001  0.121   8.00  COUNT                   BYTOP                         TESTS
010001        7  0.001  0.122   3.00  COUNT                                     ADDRS
022021        6  0.001  0.123  13.00  COUNT XJUMP                         SHIFT             TESTS
001203        6  0.001  0.124  39.00  COFIX             HALFW       LOGIC
220001        6  0.001  0.125   8.00  COUNT                                                 TESTS             INOPK
022121        6  0.001  0.126  62.00  COUNT XJUMP STORE                   SHIFT             TESTS
023107        6  0.001  0.127  17.00  CFXFL       STORE             LOGIC SHIFT             TESTS
000001        3  0.000  0.127  10.67  COUNT
004001        2  0.000  0.128   3.00  COUNT                                 STACK
022011        2  0.000  0.128  83.50  COUNT XDATA                         SHIFT             TESTS
000111        2  0.000  0.128  14.50  COUNT XDATA STORE
020013        2  0.000  0.129   6.00  COFIX XDATA                                           TESTS
000703        1  0.000  0.129  13.00  COFIX       STORE HALFW BYTOP
002101        1  0.000  0.129   5.00  COUNT       STORE                   SHIFT
020501        1  0.000  0.129  57.00  COUNT       STORE       BYTOP                         TESTS
000151        1  0.000  0.129  16.00  COUNT XIMDA STORE
020011        1  0.000  0.129  42.00  COUNT XDATA                                           TESTS

                794 LIFETIMES.   31 DIFFERENT CLASSES.

UNION CLASS AND ITS COMPLEMENT
237777                                    CFXFL XDJIM STORE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESTS             INOPK
540000                                                                                          MONIT BYTPT       EXECT



CLASS:   FULL FIXPOINT ARITHMETIC   •• • •• • •• • •• • •• • •• • •• • ••   • •• • •• • •• • •• • •• • •• • •• • •

MASK AND ITS COMPLEMENT
000002                                    FIXPT
777775                                    COFLO XDJIM STORE HALFW BYTOP LOGIC SHIFT STACK ADDRS TESTS MONIT BYTPT INOPK EXECT

                      FRAC- CUMUL.  AVRGE
 CLASS    COUNT        TION FRACT. LENGTH  INTERPRETATION

000012      150  0.024  0.024   2.52  FIXPT XDATA
020003       15  0.002  0.027  16.60  COFIX                                                 TESTS
021042       12  0.002  0.029  11.00  FIXPT XIMMC             LOGIC                         TESTS
000103       12  0.002  0.031  22.50  COFIX       STORE
022103       12  0.002  0.033   6.50  COFIX       STORE                   SHIFT             TESTS
000102       11  0.002  0.035   2.00  FIXPT       STORE
020002        9   .001  0.036  14.11  FIXPT                                                 TESTS
000603        7  0.001  0.037   8.00  COFIX             HALFW BYTOP
000002        7  0.001  0.038   2.43  FIXPT
001203        6  0.001  0.039  39.00  COFIX             HALFW       LOGIC
000112        6  0.001  0.040  52.00  FIXPT XDATA STORE
023107        6  0.001  0.041  17.00  CFXFL       STORE             LOGIC SHIFT             TESTS
023106        6  0.001  0.042  14.00  FXFLO       STORE             LOGIC SHIFT             TESTS
023102        5  0.001  0.043  29.00  FIXPT       STORE             LOGIC SHIFT             TESTS
020013        2  0.000  0.043   6.00  COFIX XDATA                                           TESTS
000703        1  0.000  0.044  13.00  COFIX       STORE HALFW BYTOP

                267 LIFETIMES.   16 DIFFERENT CLASSES.

UNION CLASS AND ITS COMPLEMENT
023757                                    CFXFL XIMDA STORE HALFW BYTOP LOGIC SHIFT             TESTS
754020                                          XJUMP                         STACK ADDRS       MONIT BYTPT INOPK EXECT
```

Output from register classification program

CLASS: FLOATING ARITHMETIC •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •• • •

```
MASK AND ITS COMPLEMENT
000004                                  FLOAT
777773                                  COFIX XOJIM STOPE HALFW BYTOP LOGIC SHIFT STACK ADDPS TESTS MONIT BYTPT INDRK EXECT
```

CLASS	COUNT	FRAC-TION	CUMUL. FRACT.	AVPGE LENGTH	INTEPPRETATION
000104	1349	0.220	0.220	5.55	FLOAT STOPE
000004	712	0.116	0.336	3.63	FLOAT
020104	20	0.003	0.339	26.00	FLOAT STORE TESTS
023104	12	0.002	0.341	79.83	FLOAT STOPE LOGIC SHIFT TESTS
002104	12	0.002	0.343	19.33	FLOAT STOPE SHIFT
000206	12	0.002	0.345	10.00	COFLO HALFW
02310[illegible]	6	0.001	0.346	17.00	CFXFL STOPE LOGIC SHIFT TESTS
023106	6	0.001	0.347	14.00	FXFLO STORE LOGIC SHIFT TESTS

2129 LIFETIMES. 8 DIFFERENT CLASSES.

```
UNION CLASS AND ITS COMPLEMENT
023307                                  CFXFL       STORE HALFW       LOGIC SHIFT             TESTS
754470                                        XDJIM             BYTOP             STACK ADDPS       MONIT BYTPT INDRK EXECT
```

CUMULATIVE STATISTICS FOR THE PHYSICAL REGISTERS

REG	LIVES	TOTAL LIVE	TOTAL USES	FRACTION LIVE	AVPG. LENGTH	USES PR. LIFE	USES PR. LIVE INSTR	USES PR. TOTAL INSTR	
00	882	5299.	3084.	0.254	6.01	3.50	0.58	0.15	
01	283	6630.	1634.	0.318	23.43	5.77	0.25	0.08	
02	2217	11590.	7940.	0.556	5.23	3.58	0.68	0.38	
03	1368	5092.	4199.	0.244	3.72	3.07	0.82	0.20	
04	298	1262.	783.	0.060	4.23	2.63	0.62	0.04	
05	21	4858.	231.	0.233	231.33	11.00	0.05	0.01	
06	48	684.	72.	0.033	14.25	1.50	0.11	0.00	
07	8	5058.	369.	0.242	632.25	46.12	0.07	0.02	
10	0	0.	0.	0.000	0.00	0.00	0.00	0.00	
11	8	5691.	517.	0.273	711.37	64.62	0.09	0.02	
12	0	0.	0.	0.000	0.00	0.00	0.00	0.00	
13	12	3462.	348.	0.166	288.50	29.00	0.10	0.02	
14	12	1044.	48.	0.050	87.00	4.00	0.05	0.00	
15	316	11376.	5122.	0.545	36.00	16.21	0.45	0.25	
16	626	9346.	2705.	0.448	14.93	4.32	0.29	0.13	
17	34	7427.	1168.	0.356	218.44	34.35	0.16	0.06	
SUM OR AVERAGES:	6133			3.779	12.85	4.60	0.36	0.09	1.35

UNION OF USAGE CLASSES FOR THE PHYSICAL REGISTERS:

```
00  637707                              CFXFL       STOPE HALFW BYTOP LOGIC SHIFT STACK ADDPS TESTS             INDRK EXECT
01  033757                              CFXFL XIMDA STOPE HALFW BYTOP LOGIC SHIFT       ADDRS TESTS
02  023537                              CFXFL XDAJM STOPE       BYTOP LOGIC SHIFT             TESTS
03  023537                              CFXFL XDAJM STORE       BYTOP LOGIC SHIFT             TESTS
04  021557                              CFXFL XIMDA STOPE       BYTOP LOGIC                   TESTS
05  021310                                    XDATA STOPE HALFW       LOGIC                   TESTS
06  020100                                          STOPE                                     TESTS
07  002610                                    XDATA       HALFW BYTOP       SHIFT
10  000000
11  021320                                    XJUMP STOPE HALFW       LOGIC                   TESTS
12  000000
13  320350                                    XIMDA STOPE HALFW                               TESTS       BYTPT INDRK
14  020001                              COUNT                                                 TESTS
15  020351                              COUNT XIMDA STORE HALFW                               TESTS
16  021170                                    XDJIM STOPE             LOGIC                   TESTS
17  014110                                    XDATA STORE                         STACK ADDPS

UNION OF CLASSES AND COMPLEMENT
737777                                  CFXFL XDJIM STORE HALFW BYTOP LOGIC SHIFT STACK ADDPS TESTS       BYTPT INDRK EXECT
040000                                                                                              MONIT
```

APPENDIX D

The total SNIFT

TOTAL EXECUTED INSTRUCTIONS AND TIME: 1000005 3211889.59 USEC.

274 DIFFERENT INSTRUCTIONS USED

THE SNIFT ORDERED BY NUMERIC OPCODE.
WITH INSTRUCTION COUNT AND COMPUTED TIME.

00	000	0	0.00	001	0	0.00	002	2	0.00	003	2	0.00	004	308	0.00	005	1068	0.00	006	422	0.00	007	449	0.00
01	010	56	0.00	011	89	0.00	012	34	0.00	013	0	0.00	014	0	0.00	015	4	0.00	016	2	0.00	017	11	0.00
02	020	3	0.00	021	4	0.00	022	0	0.00	023	0	0.00	024	0	0.00	025	0	0.00	026	0	0.00	027	0	0.00
03	030	0	0.00	031	0	0.00	032	0	0.00	033	0	0.00	034	0	0.00	035	2	0.00	036	0	0.00	037	0	0.00
04	040	6	0.00	041	4	0.00	042	0	0.00	043	0	0.00	044	0	0.00	045	0	0.00	046	0	0.00	047	137	0.00
05	050	-4	0.00	051	576	0.00	052	0	0.00	053	0	0.00	054	0	0.00	055	0	0.00	056	2	0.00	057	1	0.00
06	060	0	0.00	061	3	0.00	062	27	0.00	063	3	0.00	064	3	0.00	065	3	0.00	066	5	0.00	067	40	0.00
07	070	15	0.00	071	14	0.00	072	0	0.00	073	0	0.00	074	0	0.00	075	0	0.00	076	2	0.00	077	0	0.00
10	100	0	0.00	101	0	0.00	102	0	0.00	103	0	0.00	104	0	0.00	105	0	0.00	106	0	0.00	107	0	0.00
11	110	0	0.00	111	0	0.00	112	0	0.00	113	0	0.00	114	0	0.00	115	0	0.00	116	0	0.00	117	0	0.00
12	120	0	0.00	121	0	0.00	122	0	0.00	123	0	0.00	124	0	0.00	125	0	0.00	126	0	0.00	127	0	0.00
13	UFA	2558	12764.42	DFN	3	10.62	FSC	7886	85641.96	IBP	241	730.23	ILDB	3601	28519.92	LDB	6212	47521.80	IDPB	1914	16958.04	DPB	821	6978.50
14	FAD	10796	54843.68	FADI	2786	13555.98	FADM	546	3303.30	FADB	86	520.30	FADR	11353	61987.38	FADRI	1250	5737.50	FADRM	3982	25604.26	FADRB	2722	17502.46
15	FSB	287	1509.62	FSBI	0	0.00	FSBM	0	0.00	FSBB	0	0.00	FSBR	12876	72620.64	FSBRI	462	2203.74	FSBRM	236	1559.96	FSBRB	0	0.00
16	FMP	4173	43858.23	FMPI	272	3089.92	FMPM	0	0.00	FMPB	0	0.00	FMPR	19386	213052.14	FMPRI	1143	10229.85	FMPRM	158	1809.68	FMPRB	512	6123.52
17	FDV	5034	71986.20	FDVI	1	15.80	FDVM	86	1315.80	FDVB	0	0.00	FDVR	5533	79121.90	FDVRI	321	4301.40	FDVRM	5	76.50	FDVRB	33	504.90
20	MOVE	191789	466047.27	MOVEI	36075	53030.25	MOVEM	72293	186515.94	MOVES	0	0.00	MOVS	949	2306.07	MOVSI	3290	4836.30	MOVSM	556	1434.48	MOVSS	12	34.44
21	MOVN	5097	13303.17	MOVNI	2013	3321.45	MOVNM	1138	3140.88	MOVNS	407	1241.35	MOVM	2519	6574.59	MOVMI	0	0.00	MOVMM	0	0.00	MOVMS	315	960.75
22	IMUL	6513	63892.53	IMULI	3983	32660.60	IMULM	220	2371.60	IMULB	25	269.50	MUL	117	1265.94	MULI	175	1520.75	MULM	0	0.00	MULB	0	0.00
23	IDIV	2182	36439.40	IDIVI	2581	40779.80	IDIVM	0	0.00	IDIVB	0	0.00	DIV	0	0.00	DIVI	0	0.00	DIVM	0	0.00	DIVB	0	0.00
24	ASH	15230	36552.00	ROT	3345	8028.00	LSH	7630	18312.00	JFFO	1208	4711.20	ASHC	2070	9956.70	ROTC	1752	8427.12	LSHC	1111	5343.91	NULL	0	0.00
25	EXCH	1737	5228.37	BLT	570	26425.20	AOBJP	1277	2285.83	AOBJN	709	1269.11	JRST	70180	103164.60	JFCL	2390	3513.30	XCT	5168	7596.96	NULL	0	0.00
26	PUSHJ	18205	56004.15	PUSH	30236	123060.52	POP	13954	57909.10	POPJ	21050	66939.00	JSR	3251	9070.29	JSP	4759	6995.73	JSA	2812	8239.16	JRA	2836	8905.04
27	ADD	79290	218047.50	ADDI	11394	20395.26	ADDM	1180	3764.20	ADDB	287	915.53	SUB	11346	31201.50	SUBI	4337	7763.23	SUBM	11	35.09	SUBB	0	0.00
30	CAI	74	132.46	CAIL	726	1299.54	CAIE	4246	7600.34	CAILE	2825	5056.75	CAIA	1504	2692.16	CAIGE	1812	3243.48	CAIN	7186	12862.94	CAIG	3706	6633.74
31	CAM	0	0.00	CAML	6089	18944.75	CAME	4627	12724.25	CAMLE	14466	39781.50	CAMA	0	0.00	CAMGE	11240	30910.00	CAMN	1516	4169.00	CAMG	5783	15903.25

32	JUMP	14	JUMPL	3822	JUMPE	5818	JUMPLE	4178	JUMPA	0	JUMPGE	1431	JUMPN	3703	JUMPG	1147
	*	25.06	*	6841.38	*	10414.22	*	7478.62	*	0.00	*	2561.49	*	6628.37	*	2053.13
33	SKIP	0	SKIPL	1222	SKIPE	2704	SKIPLE	538	SKIPA	1764	SKIPGE	3111	SKIPN	3185	SKIPG	2301
	*	0.00	*	3189.42	*	7057.44	*	1404.18	*	4604.04	*	8119.71	*	8312.85	*	6005.61
34	AOJ	2434	AOJL	5356	AOJE	125	AOJLE	21	AOJA	18297	AOJGE	1	AOJN	41	AOJG	29
	*	4356.86	*	9587.24	*	223.75	*	37.59	*	32751.63	*	1.79	*	73.39	*	51.91
35	AOS	10531	AOSL	0	AOSE	15	AOSLE	171	AOSA	122	AOSGE	949	AOSN	6	AOSG	236
	*	32119.55	*	0.00	*	45.75	*	521.55	*	372.10	*	2894.45	*	18.30	*	719.80
36	SOJ	2293	SOJL	986	SOJE	261	SOJLE	223	SOJA	233	SOJGE	2850	SOJN	158	SOJG	1841
	*	4104.47	*	1764.94	*	467.19	*	399.17	*	417.07	*	5101.50	*	282.82	*	3295.39
37	SOS	1279	SOSL	5	SOSE	8	SOSLE	610	SOSA	4	SOSGE	44	SOSN	456	SOSG	425
	*	3900.95	*	15.25	*	24.40	*	1860.50	*	12.20	*	134.20	*	1390.80	*	1296.25
40	SETZ	3773	SETZI	66	SETZM	2440	SETZB	1760	AND	2981	ANDI	4908	ANDM	13	ANDB	2
	*	5546.31	*	97.02	*	5953.60	*	4294.40	*	7661.17	*	7901.88	*	39.13	*	6.02
41	ANDCA	19	ANDCAI	0	ANDCAM	49	ANDCAB	0	SETM	0	SETMI	0	SETMM	0	SETMB	0
	*	55.48	*	0.00	*	177.87	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00
42	ANDCM	55	ANDCMI	87	ANDCMM	0	ANDCMB	0	SETA	0	SETAI	0	SETAM	0	SETAB	0
	*	141.35	*	140.07	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00
43	XOR	137	XORI	30	XORM	4	XORB	72	IOR	702	IORI	46	IORM	117	IORB	66
	*	352.09	*	48.30	*	12.04	*	216.72	*	1804.14	*	74.06	*	352.17	*	190.66
44	ANDCB	3	ANDCBI	0	ANDCBM	0	ANDCBB	0	EQV	174	EQVI	0	EQVM	0	EQVB	0
	*	8.76	*	0.00	*	0.00	*	0.00	*	447.18	*	0.00	*	0.00	*	0.00
45	SETCA	3	SETCAI	0	SETCAM	41	SETCAB	0	ORCA	20	ORCAI	0	ORCAM	0	ORCAB	0
	*	4.83	*	0.00	*	105.78	*	0.00	*	58.40	*	0.00	*	0.00	*	0.00
46	SETCM	270	SETCMI	0	SETCMM	5	SETCMB	1	ORCM	173	ORCMI	7	ORCMM	0	ORCMB	0
	*	656.10	*	0.00	*	14.35	*	2.87	*	444.61	*	11.27	*	0.00	*	0.00
47	ORCB	0	ORCBI	0	ORCBM	0	ORCBB	0	SETO	319	SETOI	14	SETOM	305	SETOB	12
	*	0.00	*	0.00	*	0.00	*	0.00	*	468.93	*	20.58	*	744.20	*	29.28
50	HLL	738	HLLI	0	HLLM	33	HLLS	0	HRL	747	HRLI	2181	HRLM	581	HRLS	17
	*	1896.66	*	0.00	*	99.33	*	0.00	*	2181.24	*	4274.76	*	1748.81	*	48.79
51	HLLZ	1486	HLLZI	0	HLLZM	14	HLLZS	20	HRLZ	1494	HRLZI	1279	HRLZM	76	HRLZS	4
	*	3610.98	*	0.00	*	36.12	*	57.40	*	3630.42	*	1880.13	*	196.08	*	11.48
52	HLLO	10	HLLOI	0	HLLOM	0	HLLOS	0	HRLO	0	HRLOI	96	HRLOM	0	HRLOS	0
	*	24.30	*	0.00	*	0.00	*	0.00	*	0.00	*	141.12	*	0.00	*	0.00
53	HLLE	0	HLLEI	0	HLLEM	0	HLLES	0	HRLE	0	HRLEI	0	HRLEM	0	HRLES	0
	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00
54	HRR	749	HRRI	1210	HRRM	4198	HRRS	0	HLR	21	HLRI	0	HLRM	11	HLRS	32
	*	1924.93	*	1948.10	*	12635.98	*	0.00	*	61.32	*	0.00	*	33.11	*	91.84
55	HRRZ	12307	HRRZI	1084	HRRZM	782	HRRZS	367	HLRZ	9271	HLRZI	0	HLRZM	20	HLRZS	6
	*	29906.01	*	1593.48	*	2017.56	*	1053.29	*	22528.53	*	0.00	*	51.60	*	17.22
56	HRRO	16	HRROI	564	HRROM	5	HRROS	3	HLRO	0	HLROI	0	HLROM	0	HLROS	0
	*	38.88	*	829.08	*	12.90	*	8.61	*	0.00	*	0.00	*	0.00	*	0.00
57	HRRE	23	HRREI	13	HRREM	4	HRRES	5	HLRE	367	HLREI	0	HLREM	1	HLRES	0
	*	55.89	*	19.11	*	10.32	*	14.35	*	891.81	*	0.00	*	2.58	*	0.00
60	TRN	0	TLN	0	TRNE	1241	TLNE	5440	TRNA	0	TLNA	0	TRNN	7442	TLNN	2965
	*	0.00	*	0.00	*	2432.36	*	10662.40	*	0.00	*	0.00	*	14586.32	*	5811.40
61	TDN	0	TSN	0	TDNE	129	TSNE	0	TDNA	0	TSNA	0	TDNN	9	TSNN	0
	*	0.00	*	0.00	*	376.68	*	0.00	*	0.00	*	0.00	*	26.28	*	0.00
62	TRZ	288	TLZ	4344	TRZE	1417	TLZE	186	TRZA	0	TLZA	9	TRZN	59	TLZN	217
	*	564.48	*	8514.24	*	2777.32	*	364.56	*	0.00	*	17.64	*	115.64	*	425.32
63	TDZ	9	TSZ	0	TDZE	0	TSZE	0	TDZA	880	TSZA	0	TDZN	0	TSZN	0
	*	26.28	*	0.00	*	0.00	*	0.00	*	2569.60	*	0.00	*	0.00	*	0.00
64	TRC	280	TLC	1530	TRCE	0	TLCE	6	TRCA	0	TLCA	0	TRCN	0	TLCN	48
	*	548.80	*	2998.80	*	0.00	*	11.76	*	0.00	*	0.00	*	0.00	*	94.08
65	TDC	0	TSC	97	TDCE	0	TSCE	0	TDCA	0	TSCA	0	TDCN	0	TSCN	0
	*	0.00	*	283.24	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00	*	0.00
66	TRO	28	TLO	720	TROE	0	TLOE	37	TROA	2	TLOA	22	TRON	0	TLON	13
	*	54.88	*	1411.20	*	0.00	*	72.52	*	3.92	*	43.12	*	0.00	*	25.48
67	TDO	0	TSO	2	TDOE	0	TSOE	0	TDOA	18	TSOA	0	TDON	0	TSON	0
	*	0.00	*	5.84	*	0.00	*	0.00	*	52.56	*	0.00	*	0.00	*	0.00

The total SHIFT

THE GIBSON DISTRIBUTION

	CLASS	COUNT	FRACT.	TOTAL TIME	FRACT.	
1	LODST	423764	0.4238	1142426.42	0.3557	LOADS AND STORES
2	FIX+-	124382	0.1244	326604.14	0.1017	FIXED POINT ADD SUBTRACT
3	COMPA					
4	BRANC	281814	0.2818	609064.51	0.1896	BRANCHES
5	FLT+-	49443	0.0494	273723.06	0.0852	FLOATING ADD SUBTRACT
6	FLMUL	25644	0.0256	278243.34	0.0866	FLOATING MULTIPLY
7	FLDIV	11013	0.0110	157322.50	0.0490	FLOATING DIVIDE
8	FXMUL	11033	0.0110	101980.92	0.0318	FIXED MULTIPLY
9	FXDIV	4763	0.0048	77219.20	0.0240	FIXED DIVIDE
10	SHIFT	39024	0.0390	172261.69	0.0536	SHIFTS
11	LOGIC	9673	0.0097	20170.55	0.0063	LOGIC
12	MISCL	15351	0.0154	52872.61	0.0165	MISCELLANEOUS
13	INDEX					
14	FULWD					
15	I/O..	702	0.0007	0.00	0.0000	I/O INSTRUCTIONS
16	CPU..					
17	MONIT	143	0.0001	0.00	0.0000	MONITOR CALLS
18	UUUO.	3256	0.0033	0.00	0.0000	USER UUOS

THE PROGRAM STRUCTURE DISTRIBUTION

	CLASS	COUNT	FRACT.	TOTAL TIME	FRACT.	
1	MTOA	201089	0.2011	490470.52	0.1527	MOVE MEMORY TO ACC.
2	ATOM	76035	0.0760	197081.55	0.0614	MOVE ACC. TO MEMORY
3	IMTOA	41378	0.0414	61188.00	0.0191	MOVE IMMEDIATE TO ACC.
4	SETA	4172	0.0042	6132.84	0.0019	SET 0 OR -1 TO ACC.
5	SETM	4517	0.0045	11021.48	0.0034	SET 0 OR -1 TO MEMORY
6	PWTOA	43923	0.0439	154781.45	0.0482	MOVE PARTWORD TO ACC.
7	ATOPW	8460	0.0085	40700.93	0.0127	MOVE ACC. TO PARTWORD
8	BLKMV	570	0.0006	26425.20	0.0082	BLOCK MOVE
9	STBIT	7298	0.0073	14407.76	0.0045	SET BITS
10	UNUSD					
11	UNUSD					
12	ASONE	16537	0.0165	44481.83	0.0130	ADD OR SUBTRACT ONE
13	FIX+-	107845	0.1078	282122.31	0.0878	FIXED ADD SUBTRACT
14	FIX*/	15796	0.0158	179200.12	0.0558	FIXED MULTIPLY DIVIDE
15	FLOAT	86100	0.0861	709289.70	0.2200	FLOATING ARITHMETIC
16	SHIFT	39024	0.0390	172261.69	0.0536	SHIFTS
17	LOGIC	9673	0.0097	20170.55	0.0063	LOGICAL OPERATIONS
18	UNUSD					
19	UNUSD					
20	IOXFR	624	0.0006	0.00	0.0000	I/O TRANSFERS
21	IOADM	45	0.0000	0.00	0.0000	I/O ADMINISTRATION
22	UUOTH	143	0.0001	0.00	0.0000	OTHER MONITOR UUOS
23	UUUO	3256	0.0033	0.00	0.0000	USER UUOS
24	UNUSD					
25	UNUSD					
26	UNUSD					
27	SRJMP	29087	0.0291	81109.33	0.0253	SUBROUTINE JUMPS
28	SRRET	23886	0.0239	75844.04	0.0236	SUBROUTINE RETURNS
29	STKPT	44190	0.0442	180969.62	0.0563	STACKPOINTER OPERATIONS
30	AVSIM	20501	0.0205	36696.79	0.0114	TEST ACC. VERSUS IMMEDIATE
31	AVSO	20099	0.0201	35977.21	0.0112	TEST ACC. VERSUS ZERO
32	AVSME	44521	0.0445	122432.75	0.0381	TEST ACC. VERSUS MEMORY
33	MEVSO	13061	0.0131	34009.21	0.0106	TEST MEMORY VERSUS ZERO
34	BLKTS					
35	BITST	20417	0.0204	42493.32	0.0132	BIT TESTS
36	STATS	2423	0.0024	3513.30	0.0011	STATUS TESTS
37	LOOPJ	35459	0.0355	67315.07	0.0210	LOOP JUMPS
38	UNCJP	74379	0.0744	113147.64	0.0352	UNCONDITIONAL JUMPS
39	NOOPS	80	0.0001	157.52	0.0000	NO OPERATIONS
40	XCT	5160	0.0052	7596.96	0.0024	EXECUTE EFFECTIVE ADDRESS
41	MISCL	241	0.0002	730.23	0.0002	MISCELLANEOUS

MOST TIMECONSUMING INSTRUCTIONS EXCLUDING MONITOR CALLS

Relative execution time is with respect to the average instruction for this program.

	NAME	USED USEC.	FRACTION OF TOT. TIME	CUMUL. FRACTION	RELATIVE EXEC. TIME	#TIMES EXECUTED	FRACTION OF EXECNS.
1	MOVE	466047.27	0.1451	0.1451	0.7566	191789	0.1918
2	ADD	218047.50	0.0679	0.2130	0.8562	79290	0.0793
3	FMPR	213052.14	0.0663	0.2793	3.4217	19306	0.0194
4	MOVEM	186515.94	0.0581	0.3374	0.8033	72293	0.0723
5	PUSH	123060.52	0.0383	0.3757	1.2672	30236	0.0302
6	JRST	103164.60	0.0321	0.4078	0.4577	70180	0.0702
7	FSC	85641.96	0.0267	0.4345	3.3012	7806	0.0079
8	FDVR	79121.90	0.0246	0.4591	4.4522	5533	0.0055
9	FSBR	72620.64	0.0226	0.4817	1.7560	12876	0.0129
10	FDV	71986.20	0.0224	0.5041	4.4522	5034	0.0050
11	POPJ	66939.00	0.0208	0.5250	0.9901	21050	0.0210
12	IMUL	63892.53	0.0199	0.5449	3.0543	6513	0.0065
13	FADR	61987.38	0.0193	0.5642	1.6999	11353	0.0114
14	POP	57909.10	0.0180	0.5822	1.2921	13954	0.0140
15	PUSHJ	56804.15	0.0177	0.5999	0.968	18265	0.0183
16	FAD	54843.68	0.0171	0.6170	1.5016	10796	0.0108
17	MOVEI	53030.25	0.0165	0.6335	0.4577	36075	0.0361
18	LDB	47521.80	0.0148	0.6483	2.3818	6212	0.0062
19	FMP	43858.23	0.0137	0.6619	3.2722	4173	0.0042
20	IDIVI	40779.80	0.0127	0.6746	4.9192	2581	0.0026
21	CAMLE	39781.50	0.0124	0.6870	0.8562	14466	0.0145
22	ASH	36552.00	0.0114	0.6984	0.7472	15230	0.0152
23	IDIV	36439.40	0.0113	0.7097	5.1995	2182	0.0022
24	AOJA	32751.63	0.0102	0.7199	0.5573	18297	0.0183
25	IMULI	32660.60	0.0102	0.7301	2.5530	3983	0.0040
26	AOS	32119.55	0.0100	0.7401	0.9496	10531	0.0105
27	SUB	31201.50	0.0097	0.7498	0.8562	11346	0.0113
28	CAMGE	30910.00	0.0096	0.7594	0.8562	11240	0.0112
29	HRRZ	29906.01	0.0093	0.7688	0.7566	12307	0.0123
30	ILDB	28519.92	0.0089	0.7776	2.4659	3601	0.0036
31	BLT	26425.20	0.0082	0.7859	14.4339	570	0.0006
32	FADRM	25604.26	0.0080	0.7938	2.0019	3982	0.0040
33	HLRZ	22528.53	0.0070	0.8008	0.7566	9271	0.0093
34	ADDI	20395.26	0.0063	0.8072	0.5573	11394	0.0114
35	CAML	18944.75	0.0059	0.8131	0.8562	6889	0.0069
36	LSH	18312.00	0.0057	0.8188	0.7472	7630	0.0076
37	FADRB	17502.46	0.0054	0.8242	2.0019	2727	0.0027
38	IDPB	16958.04	0.0053	0.8295	2.7585	1914	0.0019
39	CAMG	15903.25	0.0050	0.8345	0.8562	5783	0.0058
40	TRNN	14586.32	0.0045	0.8390	0.6102	7442	0.0074
41	FADI	13555.98	0.0042	0.8432	1.8463	2286	0.0023
42	MOVN	13303.17	0.0041	0.8474	0.8126	5097	0.0051
43	CAIN	12862.94	0.0040	0.8514	0.5573	7186	0.0072
44	UFA	12764.42	0.0040	0.8554	1.5536	2558	0.0026
45	CAME	12724.25	0.0040	0.8593	0.8562	4627	0.0046
46	HRRM	12635.98	0.0039	0.8633	0.9371	4198	0.0042
47	TLNE	10662.40	0.0033	0.8666	0.6102	5440	0.0054
48	JUMPE	10414.22	0.0032	0.8698	0.5573	5818	0.0058
49	FMPRI	10229.85	0.0032	0.8730	2.7865	1143	0.0011
50	ASHC	9956.70	0.0031	0.8761	1.4976	2070	0.0021
51	ADJL	9587.24	0.0030	0.8791	0.5573	5356	0.0054
52	JSR	9070.29	0.0028	0.8819	0.8687	3251	0.0033
53	JRA	8905.04	0.0028	0.8847	0.9776	2836	0.0028
54	TLZ	8514.24	0.0027	0.8873	0.6102	4344	0.0043
55	ROTC	8427.17	0.0026	0.8900	1.4976	1752	0.0018
56	SKIPN	8312.85	0.0026	0.8925	0.8126	3185	0.0032
57	JSA	8239.16	0.0026	0.8951	0.9122	2812	0.0028
58	SKIPGE	8119.71	0.0025	0.8976	0.8126	3111	0.0031
59	ROT	8028.00	0.0025	0.9001	0.7472	3345	0.0033
60	ANDI	7901.88	0.0025	0.9026	0.5013	4508	0.0049
61	SUBI	7763.23	0.0024	0.9050	0.5573	4337	0.0043
62	AND	7661.17	0.0024	0.9074	0.8002	2981	0.0030
63	CAIE	7600.34	0.0024	0.9090	0.5573	4246	0.0042
64	XCT	7596.96	0.0024	0.9121	0.4577	5168	0.0052
65	JUMPLE	7478.62	0.0023	0.9145	0.5573	4178	0.0042
66	SKIPE	7057.44	0.0022	0.9167	0.8126	2704	0.0027
67	JSP	6995.73	0.0022	0.9188	0.4577	4759	0.0048
68	DPB	6978.50	0.0022	0.9210	2.6464	821	0.0008
69	JUMPL	6841.38	0.0021	0.9231	0.5573	3827	0.0038
70	CAIG	6633.74	0.0021	0.9252	0.5573	3706	0.0037
71	JUMPN	6628.37	0.0021	0.9273	0.5573	3703	0.0037
72	MOVM	6574.59	0.0020	0.9293	0.8126	2519	0.0025
73	FMPRB	6123.52	0.0019	0.9312	3.7237	512	0.0005
74	SKIPG	6005.61	0.0019	0.9331	0.8126	2301	0.0023
75	SETZM	5953.60	0.0019	0.9349	0.7597	2440	0.0024
76	TLNN	5811.40	0.0018	0.9368	0.6102	2965	0.0030
77	FADRI	5737.50	0.0018	0.9385	1.4291	1250	0.0012
78	SETZ	5546.31	0.0017	0.9403	0.4577	3773	0.0038
79	LSHC	5343.91	0.0017	0.9419	1.4976	1111	0.0011
80	EXCH	5228.37	0.0016	0.9436	0.9371	1737	0.0017
81	SOJGE	5101.50	0.0016	0.9451	0.5573	2850	0.0028
82	CAILE	5056.75	0.0016	0.9467	0.5573	2825	0.0028
83	MOVSI	4836.30	0.0015	0.9482	0.4577	3290	0.0033
84	JFFO	4711.20	0.0015	0.9497	1.2142	1200	0.0012
85	SKIPA	4604.04	0.0014	0.9511	0.8126	1764	0.0018
86	AOJ	4356.86	0.0014	0.9525	0.5573	2431	0.0024
87	FDVRI	4301.40	0.0013	0.9538	4.1720	321	0.0003
88	SETZB	4294.40	0.0013	0.9552	0.7597	1760	0.0018
89	HRLI	4274.76	0.0013	0.9565	0.6102	2181	0.0022
90	CAMN	4169.00	0.0013	0.9578	0.8562	1516	0.0015

The total SNIFT

91	SOJ	4104.47	0.0013	0.9591	0.5573	2793	0.0023
92	SOS	[illegible]00.95	0.0012	0.9603	0.9496	1279	0.0013
93	ADDM	3764.20	0.0012	0.9615	0.9932	1180	0.0012
94	HRLZ	3630.42	0.0011	0.9626	0.7566	1494	0.0015
95	HLLZ	3610.98	0.0011	0.9637	0.7566	1406	0.0015
96	JFCL	3513.30	0.0011	0.9648	0.4577	2390	0.0024
97	MOVNI	3321.45	0.0010	0.9658	0.5137	2013	0.0020
98	FADM	3303.30	0.0010	0.9669	1.8836	546	0.0005
99	SOJG	3295.39	0.0010	0.9679	0.5573	1841	0.0018
100	CAIGE	3243.48	0.0010	0.9689	0.5573	1812	0.0018
101	SKIPL	3189.42	0.0010	0.9699	0.8126	1222	0.0012
102	MOVNM	3140.88	0.0010	0.9709	0.8593	1138	0.0011
103	FMPI	3089.92	0.0010	0.9718	3.5369	272	0.0003
104	TLC	2998.80	0.0009	0.9728	0.6102	1530	0.0015
105	AOSGE	2894.45	0.0009	0.9737	0.9496	949	0.0009
106	TRZE	2777.32	0.0009	0.9745	0.6102	1417	0.0014
107	CAIA	2692.16	0.0008	0.9754	0.5573	1504	0.0015
108	TDZA	2569.60	0.0008	0.9762	0.9091	880	0.0009
109	JUMPGE	2561.49	0.0008	0.9770	0.5573	1431	0.0014
110	TRNE	2432.36	0.0008	0.9777	0.6102	1241	0.0012
111	IMULM	2371.60	0.0007	0.9785	3.3563	220	0.0002
112	MOVS	2306.07	0.0007	0.9792	0.7566	949	0.0009
113	AOBJP	2285.83	0.0007	0.9799	0.5573	1277	0.0013
114	FSBRI	2203.74	0.0007	0.9806	1.4851	462	0.0005
115	HRL	2181.24	0.0007	0.9813	0.9091	747	0.0007
116	JUMPG	2053.13	0.0006	0.9819	0.5573	1147	0.0011
117	HRRZM	2017.56	0.0006	0.9825	0.8033	782	0.0008
118	HRRI	1948.10	0.0006	0.9831	0.5013	1210	0.0012
119	HRR	1924.93	0.0006	0.9837	0.8002	749	0.0007
120	HLL	1896.66	0.0006	0.9843	0.8002	738	0.0007
121	FMPRM	1889.60	0.0006	0.9849	3.7237	158	0.0002
122	HRLZI	1880.13	0.0006	0.9855	0.4577	1279	0.0013
123	SOSLE	1860.50	0.0006	0.9861	0.9496	610	0.0006
124	IOR	1804.14	0.0006	0.9866	0.8002	702	0.0007
125	SOJL	1764.94	0.0005	0.9872	0.5573	986	0.0010
126	HRLM	1748.81	0.0005	0.9877	0.9371	581	0.0006
127	HRRZI	1593.48	0.0005	0.9882	0.4577	1084	0.0011
128	FSBRM	1559.96	0.0005	0.9887	2.0580	236	0.0002
129	MULI	1520.75	0.0005	0.9892	2.7056	175	0.0002
130	FSB	1509.62	0.0005	0.9897	1.6377	287	0.0003
131	MOVSM	1434.48	0.0004	0.9901	0.8033	556	0.0006
132	TLO	1411.2[illegible]	0.0004	0.9905	0.6102	720	0.0007
133	SKIPLE	1404.18	0.0004	0.9910	0.8126	538	0.0005
134	SOSN	1390.80	0.0004	0.9914	0.9496	456	0.0005
135	FDVM	1315.80	0.0004	0.9918	4.7636	86	0.0001
136	CAIL	1299.54	0.0004	0.9922	0.5573	726	0.0007
137	SOSG	1296.25	0.0004	0.9926	0.9496	425	0.0004
138	AOBJN	1269.11	0.0004	0.9930	0.5573	709	0.0007
139	MUL	1265.94	0.0004	0.9934	3.3688	117	0.0001
140	MOVNS	1241.35	0.0004	0.9938	0.9496	407	0.0004
141	HRRZS	1053.29	0.0003	0.9941	0.8936	367	0.0004
142	MOVMS	960.75	0.0003	0.9944	0.9496	315	0.0003
143	ADDB	915.53	0.0003	0.9947	0.9932	287	0.0003
144	HLRE	891.81	0.0003	0.9950	0.7566	367	0.0004
145	HRROI	829.08	0.0003	0.9953	0.4577	564	0.0006
146	SETOM	744.20	0.0002	0.9955	0.7597	305	0.0003
147	IBP	730.23	0.0002	0.9957	0.9134	241	0.0002
148	AOSG	719.80	0.0002	0.9959	0.9496	236	0.0002
149	SETCM	656.10	0.0002	0.9961	0.7566	270	0.0003
150	TRZ	564.48	0.0002	0.9963	0.6102	288	0.0003
151	TRC	548.80	0.0002	0.9965	0.6102	280	0.0003
152	AOSLE	521.55	0.0002	0.9966	0.9496	171	0.0002
153	FADB	520.30	0.0002	0.9968	1.8836	86	0.0001
154	FDVRB	504.90	0.0002	0.9970	4.7636	33	0.0000
155	SETO	468.93	0.0001	0.9971	0.4577	319	0.0003
156	SOJE	467.19	0.0001	0.9973	0.5573	261	0.0003
157	EQV	447.18	0.0001	0.9974	0.8002	174	0.0002
158	ORCM	444.61	0.0001	0.9975	0.8002	173	0.0002
159	TLZN	425.32	0.0001	0.9977	0.6102	217	0.0002
160	SOJA	417.07	0.0001	0.9978	0.5573	233	0.0002
161	SOJLE	399.17	0.0001	0.9979	0.5573	223	0.0002
162	TDNE	376.68	0.0001	0.9980	0.9091	129	0.0001
163	AOSA	372.10	0.0001	0.9982	0.9496	122	0.0001
164	TLZE	364.56	0.0001	0.9983	0.6102	106	0.0002
165	IORM	352.17	0.0001	0.9984	0.9371	117	0.0001
166	XOR	352.09	0.0001	0.9985	0.8002	137	0.0001
167	TSC	283.24	0.0001	0.9986	0.9091	97	0.0001
168	SOJN	282.82	0.0001	0.9987	0.5573	158	0.0002
169	IMULB	269.50	0.0001	0.9988	3.3563	25	0.0000
170	AOJE	223.75	0.0001	0.9988	0.5573	125	0.0001
171	XORB	216.72	0.0001	0.9989	0.9371	72	0.0001
172	IORB	198.66	0.0001	0.9989	0.9371	66	0.0001
173	HRLZM	196.08	0.0001	0.9990	0.8033	76	0.0001
174	ANDCAM	177.87	0.0001	0.9991	1.1302	49	0.0000
175	ANDCM	141.35	0.0000	0.9991	0.8002	55	0.0001
176	HRLOI	141.12	0.0000	0.9992	0.4577	96	0.0001
177	ANDCMI	140.07	0.0000	0.9992	0.5013	87	0.0001
178	SOSGE	134.20	0.0000	0.9992	0.9496	44	0.0000
179	CAI	132.46	0.0000	0.9993	0.5573	74	0.0001
180	TRZN	115.64	0.0000	0.9993	0.6102	59	0.0001
181	SETCAM	105.78	0.0000	0.9993	0.8033	41	0.0000
182	HLLM	99.33	0.0000	0.9994	0.9371	33	0.0000
183	SETZI	97.02	0.0000	0.9994	0.4577	66	0.0001
184	TLCN	94.08	0.0000	0.9994	0.6102	48	0.0000
185	HLRS	91.84	0.0000	0.9995	0.8936	32	0.0000
186	FDVRM	76.50	0.0000	0.9995	4.7636	5	0.0000
187	IORI	74.06	0.0000	0.9995	0.5013	46	0.0000
188	AOJN	73.39	0.0000	0.9995	0.5573	41	0.0000

The total SNIFT

189	TLOE	72.52	0.0000	0.9996	0.6102	37	0.0000
190	HLR	61.32	0.0000	0.9996	0.9091	21	0.0000
191	ORCA	58.40	0.0000	0.9996	0.9091	20	0.0000
192	HLLZS	57.40	0.0000	0.9996	0.8936	20	0.0000
193	HPPE	55.89	0.0000	0.9996	0.7566	23	0.0000
194	ANDCA	55.48	0.0000	0.9997	0.9091	19	0.0000
195	TRO	54.88	0.0000	0.9997	0.6102	18	0.0000
196	TDOA	52.56	0.0000	0.9997	0.9091	18	0.0000
197	AOJG	51.91	0.0000	0.9997	0.5573	29	0.0000
198	HLRZM	51.60	0.0000	0.9997	0.8033	20	0.0000
199	HPLS	48.79	0.0000	0.9997	0.8936	17	0.0000
200	XORI	48.30	0.0000	0.9997	0.5013	30	0.0000
201	AOSE	45.75	0.0000	0.9998	0.9496	15	0.0000
202	TLOA	43.12	0.0000	0.9998	0.6102	22	0.0000
203	ANDM	39.13	0.0000	0.9998	0.9371	13	0.0000
204	HPPO	38.88	0.0000	0.9998	0.7566	16	0.0000
205	AOJLE	37.59	0.0000	0.9998	0.5573	21	0.0000
206	HLLZM	36.12	0.0000	0.9998	0.8033	14	0.0000
207	SUBM	35.09	0.0000	0.9998	0.9932	11	0.0000
208	MOVSS	34.44	0.0000	0.9998	0.8936	12	0.0000
209	HLPM	33.11	0.0000	0.9999	0.9371	11	0.0000
210	SETOB	29.28	0.0000	0.9999	0.7597	12	0.0000
211	TOZ	26.28	0.0000	0.9999	0.9091	9	0.0000
212	TDNN	26.28	0.0000	0.9999	0.9091	9	0.0000
213	TLON	25.48	0.0000	0.9999	0.6102	13	0.0000
214	JUMP	25.06	0.0000	0.9999	0.5573	14	0.0000
215	SOSE	24.40	0.0000	0.9999	0.9496	8	0.0000
216	HLLO	24.30	0.0000	0.9999	0.7566	10	0.0000
217	SETOI	20.58	0.0000	0.9999	0.4577	14	0.0000
218	HPPEI	19.11	0.0000	0.9999	0.4577	13	0.0000
219	AOSN	18.30	0.0000	0.9999	0.9496	6	0.0000
220	TLZA	17.64	0.0000	0.9999	0.6102	9	0.0000
221	HLPZS	17.22	0.0000	0.9999	0.8936	6	0.0000
222	FOVI	15.80	0.0000	0.9999	4.9192	1	0.0000
223	SOSL	15.25	0.0000	0.9999	0.9496	5	0.0000
224	SETCMM	14.35	0.0000	1.0000	0.8936	5	0.0000
225	HPPES	14.35	0.0000	1.0000	0.8936	5	0.0000
226	HPPOM	12.90	0.0000	1.0000	0.8033	5	0.0000
227	SOSA	12.20	0.0000	1.0000	0.9496	4	0.0000
228	XORM	12.04	0.0000	1.0000	0.9371	4	0.0000
229	TLCE	11.76	0.0000	1.0000	0.6102	6	0.0000
230	HPLZS	11.48	0.0000	1.0000	0.8936	4	0.0000
231	OPCMI	11.27	0.0000	1.0000	0.5013	7	0.0000
232	DFN	10.62	0.0000	1.0000	1.1022	3	0.0000
233	HPPEM	10.32	0.0000	1.0000	0.8033	4	0.0000
234	ANDCB	8.76	0.0000	1.0000	0.9091	3	0.0000
235	HPPOS	8.61	0.0000	1.0000	0.8936	3	0.0000
236	ANDB	6.02	0.0000	1.0000	0.9371	2	0.0000
237	TSO	5.84	0.0000	1.0000	0.9091	2	0.0000
238	SETCA	4.83	0.0000	1.0000	0.5013	3	0.0000
239	TPOA	3.92	0.0000	1.0000	0.6102	2	0.0000
240	SETCMB	2.87	0.0000	1.0000	0.6936	1	0.0000
241	HLREM	2.58	0.0000	1.0000	0.8033	1	0.0000
242	AOJGE	1.79	0.0000	1.0000	0.5573	1	0.0000

MEAN EXECUTION TIME 3.21 MICROSEC.
WHICH MEANS 0.3113 MIPS.

MOST EXECUTED INSTRUCTIONS:

NAME, # TIMES EXECUTED, FRACTION, CUMUL. FRACTION

1	MOVE	191789	0.1918	0.1918
2	ADD	79290	0.0793	0.2711
3	MOVEM	72293	0.0723	0.3434
4	JRST	70180	0.0702	0.4135
5	MOVEI	36075	0.0361	0.4496
6	PUSH	30236	0.0302	0.4799
7	POPJ	21050	0.0210	0.5009
8	FMPR	19386	0.0194	0.5203
9	ADJA	18297	0.0183	0.5386
10	PUSHJ	18265	0.0183	0.5569
11	ASH	15230	0.0152	0.5721
12	CAMLE	14466	0.0145	0.5866
13	POP	13954	0.0140	0.6005
14	FSBR	12876	0.0129	0.6134
15	HPPZ	12307	0.0123	0.6257
16	ADDI	11394	0.0114	0.6371
17	FADR	11353	0.0114	0.6484
18	SUB	11346	0.0113	0.6598
19	CAMGE	11240	0.0112	0.6710
20	FAD	10796	0.0108	0.6818
21	AOS	10531	0.0105	0.6924
22	HLPZ	9271	0.0093	0.7016
23	FSC	7886	0.0079	0.7095
24	LSH	7630	0.0076	0.7171
25	TPNN	7442	0.0074	0.7246
26	CAIN	7186	0.0072	0.7318
27	CAML	6809	0.0069	0.7387
28	IMUL	6513	0.0065	0.7452
29	LDB	6212	0.0062	0.7514
30	JUMPE	5818	0.0058	0.7572
31	CAMG	5783	0.0058	0.7630
32	FDVR	5533	0.0055	0.7685
33	TLNE	5440	0.0054	0.7740
34	ADJL	5356	0.0054	0.7793
35	XCT	5168	0.0052	0.7845
36	MOVN	5097	0.0051	0.7896
37	FDV	5034	0.0050	0.7946
38	ANDI	4908	0.0049	0.7995
39	JSP	4759	0.0048	0.8043
40	CAME	4627	0.0046	0.8089
41	TLZ	4344	0.0043	0.8132
42	SUBI	4337	0.0043	0.8176
43	CAIE	4246	0.0042	0.8218
44	HPPM	4198	0.0042	0.8260
45	JUMPLE	4178	0.0042	0.8302
46	FMP	4173	0.0042	0.8344
47	IMULI	3983	0.0040	0.8384
48	FADPM	3982	0.0040	0.8423
49	JUMPL	3822	0.0038	0.8462
50	SETZ	3773	0.0038	0.8499
51	CAIG	3706	0.0037	0.8536
52	JUMPN	3703	0.0037	0.8573
53	ILDB	3601	0.0036	0.8609
54	POT	3345	0.0033	0.8643
55	MOVSI	3290	0.0033	0.8676
56	JSP	3251	0.0033	0.8708
57	SKIPN	3185	0.0032	0.8740
58	SKIPGE	3111	0.0031	0.8771
59	AND	2981	0.0030	0.8801
60	TLNN	2965	0.0030	0.8831
61	SOJGE	2850	0.0028	0.8859
62	JPA	2836	0.0028	0.8888
63	CAILE	2825	0.0028	0.8916
64	JSA	2812	0.0028	0.8944
65	FADRB	2722	0.0027	0.8971
66	SKIPE	2704	0.0027	0.8998
67	IDIVI	2581	0.0026	0.9024
68	UFA	2558	0.0026	0.9050
69	MOVM	2519	0.0025	0.9075
70	SETZM	2440	0.0024	0.9099
71	AOJ	2434	0.0024	0.9124
72	JFCL	2390	0.0024	0.9147
73	SKIPG	2301	0.0023	0.9170
74	SOJ	2293	0.0023	0.9193
75	FADI	2286	0.0023	0.9216
76	IDIV	2182	0.0022	0.9238
77	HPLI	2181	0.0022	0.9260
78	ASHC	2070	0.0021	0.9281
79	MOVNI	2013	0.0020	0.9301
80	IDPB	1914	0.0019	0.9320
81	SOS	1868	0.0019	0.9339
82	SOJG	1841	0.0018	0.9357
83	CAIGE	1812	0.0018	0.9375
84	SKIPA	1764	0.0018	0.9393
85	SETZB	1760	0.0018	0.9410
86	POTC	1752	0.0018	0.9428
87	EXCH	1737	0.0017	0.9445
88	TLC	1530	0.0015	0.9461
89	CAMN	1516	0.0015	0.9476
90	CAIA	1504	0.0015	0.9491
91	HPLZ	1494	0.0015	0.9506
92	HLLZ	1486	0.0015	0.9521
93	JUMPGE	1431	0.0014	0.9535
94	TRZE	1417	0.0014	0.9549

95	HPLZI	1279	0.0013	0.9562
96	SOS	1279	0.0013	0.9575
97	AOBJP	1277	0.0013	0.9587
98	FADPI	1250	0.0012	0.9600
99	TPNE	1241	0.0012	0.9612
100	SKIPL	1222	0.0012	0.9624
101	HPRI	1210	0.0012	0.9637
102	JFFO	1208	0.0012	0.9649
103	ADDM	1180	0.0012	0.9660
104	JUMPG	1147	0.0011	0.9672
105	FMPPI	1143	0.0011	0.9683
106	MOVNM	1138	0.0011	0.9695
107	LSHC	1111	0.0011	0.9706
108	HPPZI	1084	0.0011	0.9717
109	SOJL	986	0.0010	0.9727
110	MOVS	949	0.0009	0.9736
111	AOSGE	949	0.0009	0.9746
112	TDZA	880	0.0009	0.9754
113	DPB	821	0.0008	0.9763
114	HPRZM	782	0.0008	0.9770
115	HRR	749	0.0007	0.9778
116	HPL	747	0.0007	0.9785
117	HLL	738	0.0007	0.9793
118	CAIL	726	0.0007	0.9800
119	TLO	720	0.0007	0.9807
120	AOBJN	709	0.0007	0.9814
121	IOP	702	0.0007	0.9821
122	SOSLE	610	0.0006	0.9827
123	HPLM	581	0.0006	0.9833
124	051	576	0.0006	0.9839
125	BLT	570	0.0006	0.9845
126	HPPOI	564	0.0006	0.9850
127	MOVSM	556	0.0006	0.9856
128	FADM	546	0.0005	0.9861
129	SKIPLE	538	0.0005	0.9867
130	FMPPB	512	0.0005	0.9872
131	FSBPI	462	0.0005	0.9876
132	SOSN	456	0.0005	0.9881
133	007	449	0.0004	0.9885
134	SOSG	425	0.0004	0.9890
135	006	422	0.0004	0.9894
136	MOVNS	407	0.0004	0.9898
137	HPPZS	367	0.0004	0.9902
138	HLPE	367	0.0004	0.9905
139	FDVPI	321	0.0003	0.9909
140	SETO	319	0.0003	0.9912
141	MOVMS	315	0.0003	0.9915
142	004	308	0.0003	0.9918
143	SETOM	305	0.0003	0.9921
144	TPZ	288	0.0003	0.9924
145	ADDB	287	0.0003	0.9927
146	FSB	287	0.0003	0.9930
147	TPC	280	0.0003	0.9932
148	FMPI	272	0.0003	0.9935
149	SETCM	270	0.0003	0.9938
150	SOJE	261	0.0003	0.9940
151	IOP	241	0.0002	0.9943
152	FSBPM	236	0.0002	0.9945
153	AOSG	236	0.0002	0.9948
154	SOJA	233	0.0002	0.9950
155	SOJLE	223	0.0002	0.9952
156	IMULM	220	0.0002	0.9954
157	TLZN	217	0.0002	0.9957
158	TLZE	186	0.0002	0.9958
159	MULI	175	0.0002	0.9960
160	EQV	174	0.0002	0.9962
161	OPCM	173	0.0002	0.9964
162	AOSLE	171	0.0002	0.9965
163	FMPPM	158	0.0002	0.9967
164	SOJN	158	0.0002	0.9968
165	047	137	0.0001	0.9970
166	XOP	137	0.0001	0.9971
167	TDNE	129	0.0001	0.9973
168	AOJE	125	0.0001	0.9974
169	AOSA	122	0.0001	0.9975
170	MUL	117	0.0001	0.9976
171	IOPM	117	0.0001	0.9977
172	TSC	97	0.0001	0.9978
173	HPLOI	96	0.0001	0.9979
174	011	89	0.0001	0.9980
175	ANDCMI	87	0.0001	0.9981
176	FADB	86	0.0001	0.9982
177	FDVM	86	0.0001	0.9983
178	HPLZM	76	0.0001	0.9983
179	CAI	74	0.0001	0.9984
180	XOPB	72	0.0001	0.9985
181	IOPB	66	0.0001	0.9986
182	SETZI	66	0.0001	0.9986
183	TPZN	59	0.0001	0.9987
184	010	56	0.0001	0.9987
185	ANDCM	55	0.0001	0.9988
186	ANDCAM	49	0.0000	0.9988
187	TLCN	48	0.0000	0.9989
188	IOPI	46	0.0000	0.9989
189	SOSGE	44	0.0000	0.9990
190	SETCAM	41	0.0000	0.9990
191	AOJN	41	0.0000	0.9991
192	067	40	0.0000	0.9991

The total SNIFT

193	TLDE	37	0.0000	0.9991
194	012	34	0.0000	0.9992
195	HLLM	33	0.0000	0.9992
196	FDVRB	33	0.0000	0.9992
197	HLPS	32	0.0000	0.9993
198	XDP1	30	0.0000	0.9993
199	ADJG	29	0.0000	0.9993
200	TPD	28	0.0000	0.9994
201	062	27	0.0000	0.9994
202	IMULB	25	0.0000	0.9994
203	HPPE	23	0.0000	0.9994
204	TLDA	22	0.0000	0.9995
205	ADJLE	21	0.0000	0.9995
206	HLR	21	0.0000	0.9995
207	DPCA	20	0.0000	0.9995
208	HLLZS	20	0.0000	0.9995
209	HLPZM	20	0.0000	0.9996
210	ANDCA	19	0.0000	0.9996
211	TDDA	18	0.0000	0.9996
212	HPLS	17	0.0000	0.9996
213	HPPD	16	0.0000	0.9996
214	070	15	0.0000	0.9996
215	ADSE	15	0.0000	0.9997
216	071	14	0.0000	0.9997
217	HLLZM	14	0.0000	0.9997
218	SETD1	14	0.0000	0.9997
219	JUMP	14	0.0000	0.9997
220	HPPEI	13	0.0000	0.9997
221	TLDN	13	0.0000	0.9997
222	ANDM	13	0.0000	0.9998
223	MOVSS	12	0.0000	0.9998
224	SETDB	12	0.0000	0.9998
225	SUBM	11	0.0000	0.9998
226	HLRM	11	0.0000	0.9998
227	017	11	0.0000	0.9998
228	HLLD	10	0.0000	0.9998
229	TDNN	9	0.0000	0.9998
230	TDZ	9	0.0000	0.9998
231	TLZA	9	0.0000	0.9998
232	SOSE	8	0.0000	0.9999
233	DPCMI	7	0.0000	0.9999
234	AOSN	6	0.0000	0.9999
235	TLCE	6	0.0000	0.9999
236	040	6	0.0000	0.9999
237	HLPZS	6	0.0000	0.9999
238	HPPDM	5	0.0000	0.9999
239	SOSL	5	0.0000	0.9999
240	066	5	0.0000	0.9999
241	HPPES	5	0.0000	0.9999
242	SETCMM	5	0.0000	0.9999
243	FDVRM	5	0.0000	0.9999
244	041	4	0.0000	0.9999
245	SOSA	4	0.0000	0.9999
246	021	4	0.0000	0.9999
247	015	4	0.0000	0.9999
248	050	4	0.0000	0.9999
249	HPPEM	4	0.0000	0.9999
250	XDPM	4	0.0000	0.9999
251	HPLZS	4	0.0000	0.9999
252	HPPDS	3	0.0000	1.0000
253	DFN	3	0.0000	1.0000
254	061	3	0.0000	1.0000
255	ANDCB	3	0.0000	1.0000
256	064	3	0.0000	1.0000
257	020	3	0.0000	1.0000
258	065	3	0.0000	1.0000
259	SETCA	3	0.0000	1.0000
260	063	3	0.0000	1.0000
261	076	2	0.0000	1.0000
262	002	2	0.0000	1.0000
263	035	2	0.0000	1.0000
264	ANDB	2	0.0000	1.0000
265	056	2	0.0000	1.0000
266	016	2	0.0000	1.0000
267	003	2	0.0000	1.0000
268	TPDA	2	0.0000	1.0000
269	TSD	2	0.0000	1.0000
270	FDV1	1	0.0000	1.0000
271	HLPEM	1	0.0000	1.0000
272	ADJGE	1	0.0000	1.0000
273	SETCMB	1	0.0000	1.0000
274	057	1	0.0000	1.0000

The total SNIFT

INSTRUCTION SET UTILISATION

INFORMATION THEORETICAL:

BY # EXECUTED INSTRUCTIONS. ACTUAL: 5.4816
BY EXECUTION TIMES. ACTUAL: 5.6289
THEORETICAL MAXIMUM: 8.7245

FOSTER-GONTER-RISEMAN FUNCTION

#OPCODES USED	#OPCODES RECODED	#OPCODES INTERP.	FRACTION INTERP.	% INCR. IN # EXECUTED INSTR. & TIME RECODING FACTORS ARE 2.	4.	8.	16.
274	0	0	0.0000	0.0000	0.0000	0.0000	0.0000
273	1	1	0.0000	0.0000	0.0000	0.0000	0.0000
272	2	2	0.0000	0.0000	0.0000	0.0000	0.0000
271	3	3	0.0000	0.0000	0.0000	0.0000	0.0000
270	4	4	0.0000	0.0000	0.0000	0.0000	0.0001
269	5	5	0.0000	0.0000	0.0000	0.0000	0.0001
268	6	7	0.0000	0.0000	0.0000	0.0001	0.0001
267	7	9	0.0000	0.0000	0.0000	0.0001	0.0001
266	8	11	0.0000	0.0000	0.0000	0.0001	0.0002
265	9	13	0.0000	0.0000	0.0001	0.0001	0.0002
264	10	15	0.0000	0.0000	0.0001	0.0001	0.0002
263	11	17	0.0000	0.0000	0.0001	0.0001	0.0003
262	12	19	0.0000	0.0000	0.0001	0.0002	0.0003
261	13	21	0.0000	0.0000	0.0001	0.0002	0.0003
260	14	23	0.0000	0.0000	0.0001	0.0002	0.0004
259	15	26	0.0000	0.0001	0.0001	0.0002	0.0004
258	16	29	0.0000	0.0001	0.0001	0.0002	0.0005
257	17	32	0.0000	0.0001	0.0001	0.0003	0.0005
255	18	35	0.0000	0.0001	0.0001	0.0003	0.0006
255	19	38	0.0000	0.0001	0.0002	0.0003	0.0006
254	20	41	0.0000	0.0001	0.0002	0.0003	0.0007
253	21	44	0.0000	0.0001	0.0002	0.0004	0.0007
252	22	47	0.0000	0.0001	0.0002	0.0004	0.0008
251	23	50	0.0000	0.0001	0.0002	0.0004	0.0008
250	24	54	0.0001	0.0001	0.0002	0.0004	0.0009
219	25	58	0.0001	0.0001	0.0002	0.0005	0.0009
248	26	62	0.0001	0.0001	0.0002	0.0005	0.0010
247	27	66	0.0001	0.0001	0.0003	0.0005	0.0011
246	28	70	0.0001	0.0001	0.0003	0.0006	0.0011
245	29	74	0.0001	0.0001	0.0003	0.0006	0.0012
244	30	78	0.0001	0.0002	0.0003	0.0006	0.0012
243	31	82	0.0001	0.0002	0.0003	0.0007	0.0013
242	32	87	0.0001	0.0002	0.0003	0.0007	0.0014
241	33	92	0.0001	0.0002	0.0004	0.0007	0.0015
240	34	97	0.0001	0.0002	0.0004	0.0008	0.0016
239	35	102	0.0001	0.0002	0.0004	0.0008	0.0016
238	36	107	0.0001	0.0002	0.0004	0.0009	0.0017
237	37	112	0.0001	0.0002	0.0004	0.0009	0.0018
236	38	118	0.0001	0.0002	0.0005	0.0009	0.0019
235	39	124	0.0001	0.0002	0.0005	0.0010	0.0020
234	40	130	0.0001	0.0003	0.0005	0.0010	0.0021
233	41	136	0.0001	0.0003	0.0005	0.0011	0.0022
232	42	143	0.0001	0.0003	0.0006	0.0011	0.0023
231	43	151	0.0002	0.0003	0.0006	0.0012	0.0024
230	44	160	0.0002	0.0003	0.0006	0.0013	0.0026
229	45	169	0.0002	0.0003	0.0007	0.0014	0.0027
228	46	178	0.0002	0.0004	0.0007	0.0014	0.0028
227	47	188	0.0002	0.0004	0.0008	0.0015	0.0030
226	48	199	0.0002	0.0004	0.0008	0.0016	0.0032
225	49	210	0.0002	0.0004	0.0008	0.0017	0.0034
224	50	221	0.0002	0.0004	0.0009	0.0018	0.0035
223	51	233	0.0002	0.0005	0.0009	0.0019	0.0037
222	52	245	0.0002	0.0005	0.0010	0.0020	0.0039
221	53	258	0.0003	0.0005	0.0010	0.0021	0.0041
220	54	271	0.0003	0.0005	0.0011	0.0022	0.0043
219	55	284	0.0003	0.0006	0.0011	0.0023	0.0045
218	56	298	0.0003	0.0006	0.0012	0.0024	0.0048
217	57	312	0.0003	0.0006	0.0012	0.0025	0.0050
216	58	326	0.0003	0.0007	0.0013	0.0026	0.0052
215	59	340	0.0003	0.0007	0.0014	0.0027	0.0054
214	60	355	0.0004	0.0007	0.0014	0.0028	0.0057
213	61	370	0.0004	0.0007	0.0015	0.0030	0.0059
212	62	386	0.0004	0.0008	0.0015	0.0031	0.0062
211	63	403	0.0004	0.0008	0.0016	0.0032	0.0064
210	64	421	0.0004	0.0008	0.0017	0.0034	0.0067
209	65	440	0.0004	0.0009	0.0018	0.0035	0.0070
208	66	460	0.0005	0.0009	0.0018	0.0037	0.0074
207	67	480	0.0005	0.0010	0.0019	0.0038	0.0077
206	68	500	0.0005	0.0010	0.0020	0.0040	0.0080
205	69	521	0.0005	0.0010	0.0021	0.0042	0.0083
204	70	542	0.0005	0.0011	0.0022	0.0043	0.0087
203	71	564	0.0006	0.0011	0.0023	0.0045	0.0090
202	72	587	0.0006	0.0012	0.0023	0.0047	0.0094
201	73	612	0.0006	0.0012	0.0024	0.0049	0.0098
200	74	639	0.0006	0.0013	0.0026	0.0051	0.0102
199	75	667	0.0007	0.0013	0.0027	0.0053	0.0107
198	76	696	0.0007	0.0014	0.0028	0.0056	0.0111
197	77	726	0.0007	0.0015	0.0029	0.0058	0.0116
196	78	758	0.0008	0.0015	0.0030	0.0061	0.0121
195	79	791	0.0008	0.0016	0.0032	0.0063	0.0127
154	80	824	0.0008	0.0016	0.0033	0.0066	0.0132
193	81	858	0.0009	0.0017	0.0034	0.0069	0.0137
192	82	895	0.0009	0.0018	0.0036	0.0072	0.0143
191	83	935	0.0009	0.0019	0.0037	0.0075	0.0150

190	84	976	0.0010	0.0020	0.0039	0.0078	0.0156
189	85	1017	0.0010	0.0020	0.0041	0.0081	0.0163
188	86	1061	0.0011	0.0021	0.0042	0.0085	0.0170
187	87	1107	0.0011	0.0022	0.0044	0.0089	0.0177
186	88	1155	0.0012	0.0023	0.0046	0.0092	0.0185
185	89	1204	0.0012	0.0024	0.0048	0.0096	0.0193
184	90	1259	0.0013	0.0025	0.0050	0.0101	0.0201
183	91	1315	0.0013	0.0026	0.0053	0.0105	0.0210
182	92	1374	0.0014	0.0027	0.0055	0.0110	0.0220
181	93	1440	0.0014	0.0029	0.0058	0.0115	0.0230
180	94	1506	0.0015	0.0030	0.0060	0.0120	0.0241
179	95	1578	0.0016	0.0032	0.0063	0.0126	0.0252
178	96	1657	0.0017	0.0033	0.0066	0.0132	0.0264
177	97	1728	0.0017	0.0035	0.0069	0.0138	0.0276
176	98	1814	0.0018	0.0036	0.0073	0.0145	0.0290
175	99	1900	0.0019	0.0038	0.0076	0.0152	0.0304
174	100	1987	0.0020	0.0040	0.0079	0.0159	0.0318
173	101	2076	0.0021	0.0042	0.0083	0.0166	0.0332
172	102	2172	0.0022	0.0043	0.0087	0.0174	0.0348
171	103	2269	0.0023	0.0045	0.0091	0.0182	0.0363
170	104	2386	0.0024	0.0048	0.0095	0.0191	0.0382
169	105	2503	0.0025	0.0050	0.0100	0.0200	0.0400
168	106	2625	0.0026	0.0052	0.0105	0.0210	0.0420
167	107	2750	0.0027	0.0055	0.0110	0.0220	0.0440
166	108	2879	0.0029	0.0058	0.0115	0.0230	0.0461
165	109	3016	0.0030	0.0060	0.0121	0.0241	0.0483
164	110	3153	0.0032	0.0063	0.0126	0.0252	0.0504
163	111	3311	0.0033	0.0066	0.0132	0.0265	0.0530
162	112	3469	0.0035	0.0069	0.0139	0.0278	0.0555
161	113	3640	0.0036	0.0073	0.0146	0.0291	0.0582
160	114	3813	0.0038	0.0076	0.0153	0.0305	0.0610
159	115	3987	0.0040	0.0080	0.0159	0.0319	0.0638
158	116	4162	0.0042	0.0083	0.0166	0.0333	0.0666
157	117	4348	0.0043	0.0087	0.0174	0.0348	0.0696
156	118	4565	0.0046	0.0091	0.0183	0.0365	0.0730
155	119	4785	0.0048	0.0096	0.0191	0.0383	0.0766
154	120	5008	0.0050	0.0100	0.0200	0.0401	0.0801
153	121	5241	0.0052	0.0105	0.0210	0.0419	0.0839
152	122	5477	0.0055	0.0110	0.0219	0.0438	0.0876
151	123	5713	0.0057	0.0114	0.0229	0.0457	0.0914
150	124	5954	0.0060	0.0119	0.0238	0.0476	0.0953
149	125	6215	0.0062	0.0124	0.0249	0.0497	0.0994
148	126	6485	0.0065	0.0130	0.0259	0.0519	0.1038
147	127	6757	0.0068	0.0135	0.0270	0.0541	0.1081
146	128	7037	0.0070	0.0141	0.0281	0.0563	0.1126
145	129	7324	0.0073	0.0146	0.0293	0.0586	0.1172
144	130	7611	0.0076	0.0152	0.0304	0.0609	0.1218
143	131	7899	0.0079	0.0158	0.0316	0.0632	0.1264
142	132	8204	0.0082	0.0164	0.0328	0.0656	0.1313
141	133	8512	0.0085	0.0170	0.0340	0.0681	0.1362
140	134	8827	0.0088	0.0177	0.0353	0.0706	0.1412
139	135	9146	0.0091	0.0183	0.0366	0.0732	0.1463
138	136	9467	0.0095	0.0189	0.0379	0.0757	0.1515
137	137	9834	0.0098	0.0197	0.0393	0.0787	0.1573
136	138	10201	0.0102	0.0204	0.0408	0.0816	0.1632
135	139	10608	0.0106	0.0212	0.0424	0.0849	0.1697
134	140	11030	0.0110	0.0221	0.0441	0.0882	0.1765
133	141	11455	0.0115	0.0229	0.0458	0.0916	0.1833
132	142	11904	0.0119	0.0238	0.0476	0.0952	0.1905
131	143	12360	0.0124	0.0247	0.0494	0.0989	0.1978
130	144	12822	0.0128	0.0256	0.0513	0.1026	0.2052
129	145	13334	0.0133	0.0267	0.0533	0.1067	0.2133
128	146	13872	0.0139	0.0277	0.0555	0.1110	0.2220
127	147	14418	0.0144	0.0288	0.0577	0.1153	0.2307
126	148	14974	0.0150	0.0299	0.0599	0.1198	0.2396
125	149	15538	0.0155	0.0311	0.0622	0.1243	0.2486
124	150	16108	0.0161	0.0322	0.0644	0.1289	0.2577
123	151	16681	0.0167	0.0334	0.0667	0.1335	0.2669
122	152	17265	0.0173	0.0345	0.0691	0.1381	0.2762
121	153	17875	0.0179	0.0357	0.0715	0.1430	0.2860
120	154	18577	0.0186	0.0372	0.0743	0.1486	0.2972
119	155	19286	0.0193	0.0386	0.0771	0.1543	0.3086
118	156	20006	0.0200	0.0400	0.0800	0.1600	0.3201
117	157	20732	0.0207	0.0415	0.0829	0.1659	0.3317
116	158	21470	0.0215	0.0429	0.0859	0.1718	0.3435
115	159	22217	0.0222	0.0444	0.0889	0.1777	0.3555
114	160	22966	0.0230	0.0459	0.0919	0.1837	0.3675
113	161	23748	0.0237	0.0475	0.0950	0.1900	0.3800
112	162	24569	0.0246	0.0491	0.0983	0.1966	0.3931
111	163	25449	0.0254	0.0509	0.1018	0.2036	0.4072
110	164	26398	0.0264	0.0528	0.1056	0.2112	0.4224
109	165	27347	0.0273	0.0547	0.1094	0.2188	0.4375
108	166	28333	0.0283	0.0567	0.1133	0.2267	0.4533
107	167	29417	0.0294	0.0588	0.1177	0.2353	0.4707
106	168	30578	0.0306	0.0611	0.1221	0.2442	0.4884
105	169	31666	0.0317	0.0633	0.1267	0.2533	0.5067
104	170	32809	0.0328	0.0656	0.1312	0.2625	0.5249
103	171	33956	0.0340	0.0679	0.1358	0.2716	0.5433
102	172	35136	0.0351	0.0703	0.1405	0.2811	0.5672
101	173	36344	0.0363	0.0727	0.1454	0.2908	0.5815
100	174	37554	0.0376	0.0751	0.1502	0.3004	0.6009
99	175	38776	0.0388	0.0776	0.1551	0.3102	0.6204
98	176	40017	0.0400	0.0800	0.1601	0.3201	0.6403
97	177	41267	0.0413	0.0825	0.1651	0.3301	0.6603
96	178	42544	0.0425	0.0851	0.1702	0.3404	0.6807
95	179	43823	0.0438	0.0876	0.1753	0.3506	0.7012
94	180	45102	0.0451	0.0902	0.1804	0.3608	0.7216
93	181	46519	0.0465	0.0930	0.1861	0.3722	0.7443

92	182	47950	0.0479	0.0959	0.1918	0.3836	0.7672
91	183	49436	0.0494	0.0989	0.1977	0.3955	0.7910
90	184	50930	0.0509	0.1019	0.2037	0.4074	0.8149
89	185	52434	0.0524	0.1049	0.2097	0.4195	0.8389
88	186	53950	0.0539	0.1079	0.2158	0.4316	0.8632
87	187	55480	0.0555	0.1110	0.2219	0.4438	0.8877
86	188	57217	0.0572	0.1144	0.2289	0.4577	0.9155
85	189	58969	0.0590	0.1179	0.2359	0.4717	0.9435
84	190	60729	0.0607	0.1215	0.2429	0.4858	0.9717
83	191	62493	0.0625	0.1250	0.2500	0.4999	0.9999
82	192	64305	0.0643	0.1286	0.2572	0.5144	1.0289
81	193	66146	0.0661	0.1323	0.2646	0.5292	1.0583
80	194	68014	0.0680	0.1360	0.2721	0.5441	1.0882
79	195	69928	0.0699	0.1399	0.2797	0.5594	1.1188
78	196	71941	0.0719	0.1439	0.2878	0.5755	1.1511
77	197	74011	0.0740	0.1480	0.2960	0.5921	1.1842
76	198	76192	0.0762	0.1524	0.3048	0.6095	1.2191
75	199	78374	0.0784	0.1567	0.3135	0.6270	1.2540
74	200	80660	0.0807	0.1613	0.3226	0.6453	1.2906
73	201	82953	0.0830	0.1659	0.3318	0.6636	1.3272
72	202	85254	0.0853	0.1705	0.3410	0.6820	1.3641
71	203	87644	0.0876	0.1753	0.3506	0.7011	1.4023
70	204	90078	0.0901	0.1802	0.3603	0.7206	1.4412
69	205	92518	0.0925	0.1850	0.3701	0.7401	1.4803
68	206	95032	0.0950	0.1901	0.3801	0.7603	1.5206
67	207	97595	0.0976	0.1952	0.3904	0.7808	1.5615
66	208	100176	0.1002	0.2004	0.4007	0.8014	1.6028
65	209	102880	0.1029	0.2058	0.4115	0.8230	1.6461
64	210	105602	0.1056	0.2112	0.4224	0.8448	1.6896
63	211	108414	0.1084	0.2168	0.4337	0.8673	1.7346
62	212	111239	0.1112	0.2225	0.4450	0.8899	1.7798
61	213	114075	0.1141	0.2281	0.4563	0.9126	1.8252
60	214	116925	0.1169	0.2338	0.4677	0.9354	1.8708
59	215	119890	0.1199	0.2398	0.4796	0.9591	1.9182
58	216	122871	0.1229	0.2457	0.4915	0.9830	1.9659
57	217	125982	0.1260	0.2520	0.5039	1.0079	2.0157
56	218	129167	0.1292	0.2583	0.5167	1.0333	2.0667
55	219	132418	0.1324	0.2648	0.5297	1.0593	2.1187
54	220	135708	0.1357	0.2714	0.5428	1.0857	2.1713
53	221	139053	0.1391	0.2781	0.5562	1.1124	2.2248
52	222	142654	0.1427	0.2853	0.5706	1.1412	2.2825
51	223	146357	0.1464	0.2927	0.5854	1.1709	2.3417
50	224	150063	0.1501	0.3001	0.6002	1.2005	2.4010
49	225	153836	0.1538	0.3077	0.6153	1.2307	2.4614
48	226	157650	0.1577	0.3153	0.6306	1.2613	2.5225
47	227	161640	0.1616	0.3233	0.6466	1.2931	2.5862
46	228	165623	0.1656	0.3312	0.6625	1.3250	2.6500
45	229	169796	0.1698	0.3396	0.6792	1.3584	2.7167
44	230	173974	0.1740	0.3479	0.6959	1.3918	2.7836
43	231	178172	0.1782	0.3563	0.7127	1.4254	2.8507
42	232	182410	0.1824	0.3648	0.7297	1.4593	2.9186
41	233	186755	0.1868	0.3735	0.7470	1.4940	2.9881
40	234	191099	0.1911	0.3822	0.7644	1.5288	3.0576
39	235	195726	0.1957	0.3915	0.7829	1.5658	3.1316
38	236	200485	0.2005	0.4010	0.8019	1.6039	3.2077
37	237	205393	0.2054	0.4108	0.8216	1.6431	3.2863
36	238	210427	0.2104	0.4209	0.8417	1.6834	3.3668
35	239	215524	0.2155	0.4310	0.8621	1.7242	3.4484
34	240	220692	0.2207	0.4414	0.8828	1.7655	3.5311
33	241	226048	0.2260	0.4521	0.9042	1.8084	3.6167
32	242	231488	0.2315	0.4630	0.9260	1.8519	3.7038
31	243	237071	0.2371	0.4741	0.9483	1.8966	3.7931
30	244	242804	0.2428	0.4856	0.9712	1.9424	3.8848
29	245	248622	0.2486	0.4972	0.9945	1.9890	3.9779
28	246	254834	0.2548	0.5097	1.0193	2.0387	4.0773
27	247	261347	0.2613	0.5227	1.0454	2.0908	4.1816
26	248	268236	0.2682	0.5365	1.0729	2.1459	4.2918
25	249	275427	0.2754	0.5509	1.1017	2.2034	4.4068
24	250	282864	0.2829	0.5657	1.1315	2.2629	4.5258
23	251	290491	0.2905	0.5810	1.1620	2.3239	4.6479
22	252	298380	0.2984	0.5968	1.1935	2.3870	4.7741
21	253	307651	0.3076	0.6153	1.2306	2.4612	4.9224
20	254	318182	0.3182	0.6364	1.2727	2.5454	5.0909
19	255	328978	0.3290	0.6580	1.3159	2.6318	5.2636
18	256	340218	0.3402	0.6804	1.3609	2.7217	5.4435
17	257	351564	0.3516	0.7031	1.4062	2.8125	5.6250
16	258	362917	0.3629	0.7258	1.4517	2.9033	5.8066
15	259	374311	0.3743	0.7486	1.4972	2.9945	5.9889
14	260	386610	0.3866	0.7732	1.5465	3.0929	6.1858
13	261	399494	0.3995	0.7990	1.5980	3.1959	6.3919
12	262	413448	0.4134	0.8269	1.6538	3.3076	6.6151
11	263	427914	0.4279	0.8558	1.7116	3.4233	6.8466
10	264	443144	0.4431	0.8863	1.7726	3.5451	7.0903
9	265	461409	0.4614	0.9228	1.8456	3.6913	7.3825
8	266	479706	0.4797	0.9594	1.9188	3.8376	7.6753
7	267	499092	0.4991	0.9982	1.9964	3.9927	7.9854
6	268	520142	0.5201	1.0403	2.0806	4.1611	8.3223
5	269	550378	0.5504	1.1008	2.2015	4.4030	8.8060
4	270	586453	0.5865	1.1729	2.3458	4.6916	9.3832
3	271	656633	0.6566	1.3133	2.6265	5.2530	10.5061
2	272	728926	0.7289	1.4578	2.9157	5.8314	11.6628
1	273	808216	0.8082	1.6164	3.2328	6.4657	12.9314

APPENDIX E

Listing of the short subject algorithms

ALGOL PROGRAMS

BAIPSTOW

```
BEGIN
COMMENT THIS IS ALGOPITHM 30 FPOM THE CACM ALGOPITHMS SECTION
TYPE-IN AND CALLING PPOGPAM BY A. LUNDE;

APPAY COEFFS[0:12],PPEAL[1:12],RIMAG[1:12],CONCON[1:12];
INTEGEP IOEG,ITEP,NOIGS,IX,ISET;

PPOCEOUPE PUTOUT(IO);
VALUE IO;  INTEGEP IO;
BEGIN
  WPITE("[ZC]DATA SET "):  PPINT(IO,3,0);
  FOP IX ← I STEP 1 UNTIL IOEG OO
  BEGIN
    WPITE("[C]");
    PPINT(PPEAL[IX],6,7);  PRINT(PIMAG[IX],6,7);
    PPINT(CONCON[IX],13,2);
  END; ! OUTPUT LOOP;
PETUPN:
END; ! PPOCEOUPE PUTOUT;

PPOCEDUPE POOTPOL(NDEG,XTCOF,LITEP,NFIGS,PPE,PIM,CONV);
VALUE NOEG,LITEP,NFIGS;
INTEGEP LITEP,NFIGS,NDEG;
APPAY XTCOF,PPE,PIM,CONV;
BEGIN
  INTEGEP I,J,M;
  APPAY COF,B,C,O,E[-2:NOEG];
  PEAL TST,ACCUP,PS,QS,PT,QT,SCL,P,PEV,P,Q;

  PPOCEDUPE PEVERSE;
  BEGIN
    TST ← -TST;
    M ← ENTIEP((NOEG-1)/2);
    FOP J ← 0 STEP I UNTIL M OO
    BEGIN
      SCL ← COF[J];  COF[J] ← COF[NOEG-J]:
      COF[NDEG-J] ← SCL;
    END; ! SWAPPING LOOP;
  ENO; ! PEVEPSE;

  INTEGEP PPOCEOUPE LINEAP;
  BEGIN
    IF TST < 0.D THEN P ← I.0/P;
    PPE[NOEG] ← P;   PIM[NDEG] ← 0.0;
    CONV[NOEG] ← ACCUP;
    NOEG ← NDEG-1:
    FOP J ← 0 STEP 1 UNTIL NOEG OD
      IF ABS(COF[J]/O[J]) < ACCUR THEN COF[J] ← O[J]
      ELSE COF[J] ← 0.0:
    LINEAP ← NOEG:
  ENO; ! PPOCEOUPE LINEAP;

  B[-1] ← B[-2] ← C[-1] ← C[-2] ← D[-1] ← E[-I] ←
  COF[-1] ← 0.0;
  FOP J ← 0 STEP 1 UNTIL NOEG OO COF[J] ← XTCOF[J];
  TST ← 1.0;  ACCUP ← 10.0↑NFIGS:

  COMMENT WHILE COF[NOEG] = 0.0 DO;
 ZPOTEST:
  IF COF[NOEG] = 0.0 THEN
  BEGIN
    PPE[NOEG] ← 0.0;   PIM[NOEG] ← 0.0;  CONV[NDEG] ← ACCUR;
    NOEG ← NOEG-I;
    GO TO ZROTEST;
  ENO;

  COMMENT UNTIL NOEG = D DO;
  BEGIN
 INIT:
    IF NOEG = 0 THEN GO TO PETUPN:
    PS ← 0.0;  QS ← 0.0;  PT ← 0.0;  QT ← 0.0;
    SCL ← 0.0;
    PEV ← 1.0;   ACCUP ← 10.0 ↑ NFIGS;

    IF NOCG = 1 THEN
    BEGIN
      P ← -COF[1]/COF[0];
      LINEAP;
      GO TO PETUPN;
    END;

    FOP J ← 0 STEP 1 UNTIL NDEG DO
    BEGIN
      IF COF[J] ≠ 0.0
        THEN SCL ← LN(ABS(COF[J]))+SCL;
    END;
    SCL ← EXP(SCL/(NOEG+1));

    FOP J ← 0 STEP I UNTIL NOEG DD COF[J] ← COF[J]/SCL;
    IF ABS(COF[1]/COF[0]) < AOS(COF[NOEG-1]/COF[NOEG])
      THEN PEVEPSE;

    COMMENT WHILE TPUE OO ! FINO LIN OR QUAD FACTOR;
    BEGIN
 PEVSED:
      IF QS ≠ 0.0 THEN
      BEGIN
        P ← PS;   Q ← QS;
      END ELSE
      BEGIN
        IF COF[NOEG-2] = 0.0 THEN
        BEGIN Q ← I.0;  P ← -2.0 END
        ELSE
        BEGIN
          Q ← COF[NOEG]/COF[NDEG-2];
          P ← (COF[NDEG-1]-Q*COF[NOEG-3])/COF[NDEG-2]
        END;
        IF NOEG = 2 THEN GO TO QAOPTIC;
        P ← 0.0;
      END;

      COMMENT WHILE TPUE OO ! LOOP FOP LINEAR FACTOP;
      BEGIN
 ITERATE:
        FOP I ← I STEP 1 UNTIL LITEP OO
        BEGIN
 BAIPSTOW:
          BEGIN
            FOP J ← 0 STEP 1 UNTIL NOEG OO
            BEGIN
              B[J] ← COF[J]-P*B[J-1]-Q*B[J-2];
              C[J] ← B[J]-P*C[J-1]-Q*C[J-2];
            END;
            IF COF[NOEG-1] ≠ 0.0 THEN
            BEGIN
              IF B[NDEG-1] ≠ 0.0 THEN
              BEGIN
                IF ABS(COF[NDEG-1]/B[NOEG-1]) < ACCUR
                  THEN GO TO NEWTON;
                B[NDEG] ← COF[NOEG]-Q*B[NOEG-2]);
              END:
            END;
 BNTEST:
            IF B[NDEG] = 0.0 THEN GO TO QAOPTIC;
            IF ABS(COF[NOEG]/B[NOEG]) > ACCUP
              THEN GO TO QAOPTIC;
          END;

 NEWTON:
          FOP J ← 0 STEP 1 UNTIL NDEG DO
          BEGIN
            D[J] ← COF[J]+P*D[J-1];
            E[J] ← D[J]+P*E[J-1];
          END;

          IF O[NOEG] = 0.0 THEN GO TO LIN;
          IF ACCUP < AOS(COF[NOEG]/D[NDEG]) THEN
          BEGIN
 LIN:
            IF LINEAP = 0 THEN GO TO PETUPN
            ELSE GO TO ITEPATE
          END;

          C[NOEG-1] ← -P*C[NOEG-2]-Q*C[NDEG-3];
          SCL ← C[NDEG-2]*C[NDEG-2]-C[NDEG-I]*C[NDEG-3];
          IF SCL = 0.0 THEN
            BEGIN P ← P-2.0;   Q ← Q*(Q+1.0);  END
          ELSE
            BEGIN
              P ← P+(B[NDEG-1]*C[NOEG-2]-B[NOEG]*C[NDEG-3])/SCL;
              Q ← Q+(-B[NDEG-1]*C[NOEG-I]+B[NDEG]*C[NOEG-2])/SCL
            END;

          IF E[NDEG-1] = 0.0 THEN P ← P-1
          ELSE P ← P-D[NDEG]/E[NOEG-1];
        END ITEPATE LOOP;
      END LINEAP FACTOP LOOP;

      PS ← PT;   QS ← QT;   PT ← P;   QT ← Q;
      IF PEV < 0.0 THEN ACCUP ← ACCUP/10.0;
      PEV ← -PEV;
      PEVEPSE;
      GO TO PEVSED;
    END FACTOP FOUND;
```

```
QHDPTIC:
    IF TST < 0.0 THEN
      BEGIN  P + P/Q;   Q + 1/Q;   END;
    IF (Q-(P/2.0)*(P/2.0)) > 0.0 THEN
    BEGIN
      PPE[NDEG] + PPE[NDEG-1] + -P/2.0;
      SCL + SQPT(Q-(P/2.0)*(P/2.0));
      PIM[NDEG] + SCL;
      PIM[NDEG-1] + -SCL;
    END ELSE
    BEGIN
      SCL + SQPT((P/2.0)*(P/2.0)-Q);
      IF P < 0.0 THEN PPE[NDEG] + -P/2.0+SCL
        ELSE PPE[NDEG] + -P/2.0-SCL;
      PPE[NDEG-1] + Q/PPE[NDEG];
      PIM[NDEG] + PIM[NDEG-1] + 0.0;
    END;
    CONV[NDEG] + ACCUP;   CONV[NDEG-1] + ACCUP;
    NDEG + NDEG-2;
    FOP J + 0 STEP 1 UNTIL NDEG DO
    BEGIN
      IF B[J] = 0.0 THEN COF[J] + 0.0
      ELSE IF ABS(COF[J]/B[J]) < ACCUP THEN COF[J] + B[J]
      ELSE COF[J] + 0.0;
    END;
    GO TO INIT;
  END; ! UNTIL NDEG = 0 DO LOOP;
 PETUPN;
 END; ! PROCEDUPE POOTPOL;


 ISET + 1;
 IDEG + 4;    ITEP + 10;    NDIGS + 7;
 COEFFS[0] + 10090000.0;  COEFFS[1] + -989130.0;
 COEFFS[2] + -109900.0;  COEFFS[3] + 100000.0;
 COEFFS[4] + 1.0;

 POOTPOL(IDEG,COEFFS,ITEP,NDIGS,RPEAL,PIMAG,CONCON);
 PUTOUT(ISET);


 ISET + 2;
 IDEG + 4;   ITEP + 10;   NDIGS + 7;
 COEFFS[0] + 1.0;    COEFFS[1] + -3.0;
 COEFFS[2] + 20.0;   COEFFS[3] + 44.0;
 COEFFS[4] + 54.0;

 POOTPOL(IDEG,COEFFS,ITER,NDIGS,PPEAL,RIMAG,CONCON);
 PUTOUT(ISET);


 ISET + 3;
 IDEG + 6;    ITEP + 40;    NDIGS + 7;
 COEFFS[0] + 1.0;  COEFFS[1] + -2.0;
 COEFFS[2] + 2.0;  COEFFS[3] + 1.0;
 COEFFS[4] + 6.0;  COEFFS[5] + -6.0;
 COEFFS[6] + 8.0;

 ROOTPOL(IDEG,COEFFS,ITEP,NDIGS,RPEAL,RIMAG,CONCON);
 PUTOUT(ISET);


 ISET + 4;
 IDEG + 5;  ITEP + 40;   NDIGS + 7;
 COEFFS[0] + 1.0;  COEFFS[1] + 1.0;
 COEFFS[2] + -8.0;  COEFFS[3] + -16.0;
 COEFFS[4] + 7.0;  COEFFS[5] + 15.0;

 ROOTPOL(IDEG,COEFFS,ITEP,NDIGS,PPEAL,RIMAG,CONCON);
 PUTOUT(ISET);


 ISET + 5;
 IDEG + 4;  ITEP + 10;   NDIGS + 7;
 COEFFS[0] + 1.0;   COEFFS[1] + 5.0;
 COEFFS [2] + 3.0;   COEFFS[3] + -5.0;
 COEFFS[4] + -9.0;

 POOTPOL(IDEG,COEFFS,ITEP,NDIGS,PPEAL,RIMAG,CONCON);
 PUTOUT(ISET);


 ISET + 6;
 IDEG + 3;   ITER + 10;   NDIGS + 7;
 COEFFS[0] + 1.0;   COEFFS[1] + -8.0;
 COEFFS [2] + 17.0;   COEFFS[3] + -10.0;

 POOTPOL(IDEG,COEFFS,ITEP,NDIGS,PPEAL,RIMAG,CONCON);
 PUTOUT(ISET);

 END
```

```
CPOUT


BEGIN
  COMMENT THIS IS CALGO ALGOPITHM 43. CPOUT LINEAR EQUNS.
    ALGOPITHM BY HENPY C. THACHER JR.,
    NEW INNEPPPODUCT POUTINE AND OTHEP DPESSINGS BY A. LUNDE
    C-MU 1972.;
  APPAY LQUAT[1:15,1:15],PIGHT[1:15],SOL[1:15];
  INTEGEP APPAY LOIAG[1:15];
  PEAL DIPMN;
  FOPWAPD LABEL SINGULAP;
  INTEGEP I,J;


  PEAL PROCEDUPE INPPR1(AL,AP,LIN,LOW,MAX);
    VALUE LIN,LOW,MAX;
    INTEGEP LIN,LOW,MAX;
    APPAY AL,AP;
  BEGIN
    LONG PEAL SUM;
    INTEGER KX;

    SUM + 0.0;
    FOP KX + LOW STEP 1 UNTIL MAX DO
         SUM + SUM+AL[LIN,KX]*AP[KX];
    INPPR1 + SUM;
  END;

  PEAL PPOCEDUPE INPPP2(APPY,LIN,KOL,LOW,MAX);
    VALUE LIN,KOL,LOW,MAX;
    INTEGEP LIN,KOL,LOW,MAX;
    APPAY APRY;
  BEGIN
    LONG PEAL SUM;
    INTEGEP KX;

    SUM + 0.0;
    FOP KX + LOW STEP 1 UNTIL MAX DO
         SUM + SUM+APPY[LIN,KX]*APPY[KX,COL];
    INPPR2 + SUM;
  END;

  PPOCEDUPE CPOUT2(APP,PHS,NBYN,PES,IVOTP,DET,PEPEAT);
    VALUE NBYN,PEPEAT;
    APPAY APP,PHS,PES;
    INTEGEP NBYN;
    INTEGEP APPAY IVOTP;
    PEAL DET;
    BOOLEAN PEPEAT;
  BEGIN
    INTEGEP IX,JX,KX,IMAX,IP;
    PEAL TEMP,QUOT;

    DET + 1.0;
    IF PEPEAT THEN GO TO LABL6;
    FOP KX + 1 STEP 1 UNTIL NBYN DO
    BEGIN
      TEMP + 0.0;
      FOP IX + KX STEP 1 UNTIL NBYN DO
      BEGIN
        APP[IX,KX] + APP[IX,KX]-INPPP2(APP,IX,KX,1,KX-1);
        IF ABS(APR[IX,KX]) > TEMP THEN
        BEGIN
          TEMP + ABS(APR[IX,KX]);
          IMAX + IX;
        END;
      END;
      IVOTP[KX] + IMAX;

      IF IMAX ≠ KX THEN
      BEGIN
        DET + - DET;
        FOP JX + 1 STEP 1 UNTIL NBYN DO
        BEGIN
          TEMP + APP[KX,JX];
          APP[KX,JX] + APP[IMAX,JX];
          APP[IMAX,JX] + TEMP;
        END;

        TEMP + PHS[KX];
        PHS[KX] + PHS[IMAX];
        PHS[IMAX] + TEMP
      END;

      IF APP[KX,KX] = 0.0 THEN GO TO SINGULAP;

      QUOT + 1.0/APP[KX,KX];
      FOR IX + KX+1 STEP 1 UNTIL NBYN DO
          APP[IX,KX] + QUOT*APP[IX,KX];
      FOP JX + KX+1 STEP 1 UNTIL NBYN DO
          APP[KX,JX] + APP[KX,JX] - INPPP2(APP,KX,JX,1,KX-1);
      PHS[KX] + PHS[KX] - INPPP1(APR,PHS,KX,1,KX-1);
    END;
    GO TO LBL7;
```

```
LABL6:  COMMENT NEW RIGHT SIDE ONLY.;
    FOR KX ← I STEP 1 UNTIL NBYN DO
    BEGIN
      TEMP ← RHS[IVOTP[KX]];
      RHS[IVOTP[KX]] ← RHS[KX];
      RHS[KX] ← TEMP;
      RHS[KX] ← RHS[KX] - INRPRI(ARR,RHS,KX,I,KX-I);
    END;

LBL7:
    FOR KX ← NBYN STEP -I UNTIL I DO
    BEGIN
      IF NOT REPEAT THEN DET ← ARR[KX,KX]*DET;
      RES[KX] ← (RHS[KX]
               - INRPRI(ARR,RES,KX,KX+I,NBYN))/ARR[KX,KX];
    END;

  END; ! THAT WAS CROUT 2.;

  FOR I ← I STEP I UNTIL IS DO
  BEGIN
    FOR J ← I STEP I UNTIL IS DO
      EQUAT[I,J] ← (I+J)/2.0;
    RIGHT[I] ← LN(I/3.0);
    EQUAT[I,I] ← EQUAT[I,I]+IS-I;
  END;

  CROUT2(EQUAT,RIGHT,IS,SOL,LDIAG,DTRMN,FALSE);

  GO TO EXIT;
  WRITE("[C]");
  PRINT(DTRMN,10,6);
  WRITE("[C]");
FOR I ← I STEP I UNTIL IS DO
BEGIN
  WRITE("[C]");
FOR J ← I STEP I UNTIL IS DO
  PRINT(EQUAT[I,J],10,6);
END;
  WRITE("[C]");
  FOR I ← I STEP I UNTIL IS DO
  PRINT(LDIAG[I],10,0);
  WRITE("[C]");
FOR I ← I STEP I UNTIL IS DO
  PRINT(RIGHT[I],10,6);
  WRITE("[C]");
FOR I ← I STEP I UNTIL IS DO
  PRINT(SOL[I],10,6);
  GO TO EXIT;

SINGULAR:
  WRITE("[C]SINGULAR[C]");

EXIT:
END;! END OF MAIN PROGRAM.;
```

TREESORT

```
BEGIN
  COMMENT ALGORITHM 113 FROM THE COLLECTED ALGORITHMS COLOUMN
    OF THE CACM.  ALGORITHM AUTHOR IS ROBERT W FLOYD.
    MAIN PROGRAM W. CALLING SEQUENCE SUPPLIED BY A. LUNDE;

  ARRAY BEFORE[1:401],AFTER[1:400];
  INTEGER INFINITY,K;

  PROCEDURE TREESORT(UNSORTED,N,SORTED,K);
    VALUE N,K;
    INTEGER N,K;
    ARRAY UNSORTED,SORTED;
  BEGIN
    INTEGER I,J;
    INTEGER ARRAY M[1:2*N-1];

    FOR I ← I STEP 1 UNTIL N DO M[N+I-1] ← I*10000+N+I-1;
    FOR I ← N-1 STEP -1 UNTIL 1 DO
      M[I] ← IF UNSORTED[M[2*I] DIV 10000]
              < UNSORTED[M[2*I+1] DIV 10000] THEN M[2*I]
           ELSE M[2*I+1];

    FOR J ← 1 STEP 1 UNTIL K DO
    BEGIN
      SORTED[J] ← UNSORTED[M[1] DIV 10000];
      I ← M[1]-(M[1] DIV 10000)*10000;
      M[I] ← INFINITY * 10000;
      FOR I ← I DIV 2 WHILE I > 0 DO
        M[I] ← IF UNSORTED[M[2*I]] DIV 10000]
                < UNSORTED[M[2*I+1] DIV 10000] THEN M[2*I]
             ELSE M[2*I+1];


    END J LOOP;

  END TREESORT;

  INFINITY ← 401;
  FOR K ← 1 STEP 1 UNTIL 400 DO BEFORE[K] ← 401.0-K;
  BEFORE[401] ← 1000D.0;

  TREESORT(BEFORE,400,AFTER,400);

  FOR K ← I STEP I UNTIL 399 DO
    IF AFTER[K] > AFTER[K+1] THEN
    BEGIN
      WRITE("[C]");
      PRINT(K,6,0);
      WRITE("    OUT OF ORDER[C]");
    END;

END MAIN PROGRAM;
```

Listing of the short subject algorithms

PERT

```
BEGIN
INTEGER NEVNTS,IX;
INTEGER ARRAY INIT,LAST,LINK[1:300];
ARRAY ESTIME,EARLYS,LATEF[1:300];
REAL TSTART;

PROCEDURE PERT(NMAX,IBEG,JEND,TE,ST,EMAX,LNK,ES,AT);
INTEGER NMAX,EMAX;
REAL ST;
INTEGER ARRAY IBEG,JEND,LNK;
REAL ARRAY TE,ES,AT;
VALUE NMAX,ST;
BEGIN

 INTEGER I;
   INTEGER NX,IEX,ISX,ITX,KX;
   REAL AXX,XXX;
   SWITCH SW2 ← G1,G2;

   PROCEDURE SCAN(TOBJ);
   INTEGER TOBJ;
   BEGIN
     INTEGER KX;
       IF IEX ≠ 1 THEN
       BEGIN
         FOR KX ← IEX-1 STEP -1 UNTIL 1 DO
           IF TOBJ = LNK[KX] THEN
           BEGIN TOBJ ← KX;  GO TO RETURN;  END
       END;
       LNK[IEX] ← TOBJ;  TOBJ ← IEX;  IEX ← IEX+1;
 RETURN:
   END SCAN;

    IEX ← 1;
    FOR NX ← 1 STEP 1 UNTIL NMAX DO
    BEGIN   SCAN(JEND[NX]);   SCAN(IBEG[NX]);   END;

   EMAX ← IEX-1;  ISX ← 1;  AXX ← ST;
  WHILETRUEDO:
    KX ← EMAX;
    FOR IEX ← 1 STEP 1 UNTIL EMAX DO AT[IEX] ← AXX;
  S2:
    FOR NX ← 1 STEP 1 UNTIL NMAX DO
    BEGIN
      IF LNK[IBEG[NX]] > 0 THEN
      BEGIN
        SWITCH SW1 ← B1,B2;
        GO TO SW1[ISX];
  B1:
        XXX ← ABS(AT[IBEG[NX]] + TE[NX]);
        IF XXX > ABS(AT[JEND[NX]]) THEN AT[JEND[NX]] ← -XXX;
        GO TO ESAC1;
  B2:
        XXX ← ABS(AT[IBEG[NX]])-TE[NX];
        IF XXX < ABS(AT[JEND[NX]]) THEN AT[JEND[NX]] ← -XXX;
   ESAC1:
      END;
    END;
    FOR IEX ← 1 STEP 1 UNTIL EMAX DO
    BEGIN
      IF LNK[IEX]< 0 THEN
      BEGIN
        IF AT[IEX] < 0 THEN
        BEGIN
          LNK[IEX] ← ABS(LNK[IEX]);   KX ← KX+1;
          AT[IEX] ← ABS(AT[IEX]);
        END;
      END ELSE
      IF AT[IEX] >= 0 THEN
        BEGIN  LNK[IEX] ← -LNK[IEX];  KX ← KX-1;  END
      ELSE AT[IEX] ← ABS(AT[IEX]);
    END;

    IF KX ≠ 0 THEN GOTO S2;
    GOTO SW2[ISX];
  G1:
    ISX ← 2;
    FOR NX ← 1 STEP 1 UNTIL NMAX DO
    BEGIN
      ITX ← IBEG[NX];   IBEG[NX] ← JEND[NX];   JEND[NX] ← ITX;
    END;
    AXX ← 0;
    FOR IEX ← 1 STEP 1 UNTIL EMAX DO
    BEGIN
      ES[IEX] ← AT[IEX];   LNK[IEX] ← ABS(LNK[IEX]);
      IF AT[IEX] > AXX THEN AXX ← AT[IEX];
    END;
    GO TO WHILETRUEDO;
  G2:
    FOR IEX ← 1 STEP 1 UNTIL EMAX DO LNK[IEX] ← ABS(LNK[IEX]);
  END PERT;

  PROCEDURE PUTOUT(NEV,LK,EAS,XLF);
  VALUE NEV;
  INTEGER NEV;
INTEGER ARRAY LK;
ARRAY EAS,XLF;
BEGIN

  WRITE("[C]");  PRINT(NEV,4,0);  WRITE("  EVENTS[C]");
  GO TO RETURN;
  FOR IX ← 1 STEP 1 UNTIL NEV DO
  BEGIN
    WRITE("[C]");  PRINT(LK[IX],4,0);
    PRINT(EAS[IX],10,4);  PRINT(XLF[IX],10,4);
    IF ABS(EAS[IX]-XLF[IX]) < 0.001
      THEN WRITE("  CRITICAL");
  END;
  WRITE("[C]");
RETURN:
END;

PROCEDURE WORK(NACTS);
VALUE NACTS;  INTEGER NACTS;
BEGIN

   INIT[1] ← 1;    LAST[1] ← 2;    ESTIME[1] ← 2.5;
   INIT[2] ← 1;    LAST[2] ← 3;    ESTIME[2] ← 1.8;
   INIT[3] ← 1;    LAST[3] ← 4;    ESTIME[3] ← 3.0;
   INIT[4] ← 1;    LAST[4] ← 10;   ESTIME[4] ← 18.4;
   INIT[5] ← 2;    LAST[5] ← 5;    ESTIME[5] ← 4.2;
   INIT[6] ← 2;    LAST[6] ← 6;    ESTIME[6] ← 3.8;
   INIT[7] ← 2;    LAST[7] ← 7;    ESTIME[7] ← 6.7;
   INIT[8] ← 3;    LAST[8] ← 6;    ESTIME[8] ← 1.1;
   INIT[9] ← 3;    LAST[9] ← 7;    ESTIME[9] ← 1.3;
   INIT[10] ← 4;   LAST[10] ← 7;   ESTIME[10] ← 0.2;
   INIT[11] ← 6;   LAST[11] ← 5;   ESTIME[11] ← 6.6;
   INIT[12] ← 6;   LAST[12] ← 8;   ESTIME[12] ← 2.2;
   INIT[13] ← 7;   LAST[13] ← 8;   ESTIME[13] ← 4.9;
   INIT[14] ← 7;   LAST[14] ← 9;   ESTIME[14] ← 3.2;
   INIT[15] ← 7;   LAST[15] ← 10;  ESTIME[15] ← 1.1;
   INIT[16] ← 5;   LAST[16] ← 11;  ESTIME[16] ← 6.0;
   INIT[17] ← 8;   LAST[17] ← 11;  ESTIME[17] ← 6.0;
   INIT[18] ← 9;   LAST[18] ← 11;  ESTIME[18] ← 8.1;
   INIT[19] ← 10;  LAST[19] ← 11;  ESTIME[19] ← 0.7;
   INIT[20] ← 5;   LAST[20] ← 12;  ESTIME[20] ← 4.8;
   INIT[21] ← 8;   LAST[21] ← 12;  ESTIME[21] ← 0.7;
   INIT[22] ← 8;   LAST[22] ← 14;  ESTIME[22] ← 6.4;
   INIT[23] ← 10;  LAST[23] ← 14;  ESTIME[23] ← 3.8;
   INIT[24] ← 12;  LAST[24] ← 13;  ESTIME[24] ← 0.2;
   INIT[25] ← 11;  LAST[25] ← 13;  ESTIME[25] ← 2.5;
   INIT[26] ← 11;  LAST[26] ← 14;  ESTIME[26] ← 0.9;
   INIT[27] ← 6;   LAST[27] ← 13;  ESTIME[27] ← 11.1;
   INIT[28] ← 7;   LAST[28] ← 5;   ESTIME[28] ← 6.0;
   INIT[29] ← 11;  LAST[29] ← 12;  ESTIME[29] ← 7.3;
   INIT[30] ← 9;   LAST[30] ← 12;  ESTIME[30] ← 3.8;
   INIT[31] ← 14;  LAST[31] ← 13;  ESTIME[31] ← 0.7;
   INIT[32] ← 4;   LAST[32] ← 10;  ESTIME[32] ← 12.6;

   TSTART ← 0.0;

   PERT(NACTS,INIT,LAST,ESTIME,TSTART,NEVNTS,LINK,EARLYS,LATEF);
   PUTOUT(NEVNTS,LINK,EARLYS,LATEF);

 END;


 WORK(32);
 WORK(27);

 END;
```

HAAVIE

```
BEGIN
  COMMENT THIS IS CALGO ALGOPITHM NO. 257, HAAVIE INTEGRATION.
  ALGOPITHM BY POBEPT N. KUBIK, PUBLISHEO CACM 1965.
  TYPED BY A. LUNDE, C-MU 1972.;

  REAL A,B,EPS,MASK,Y,ANSWER;

  REAL PROCEDURE HAVIE(A,B,EPS,GRAND,M);
    VALUE A,B,EPS,M;
    INTEGER M;
    REAL A,B,EPS;
    REAL PROCEDURE GRAND;

  BEGIN
    REAL H,ENDPTS,SUMT,SUMU,D,X;
    INTEGER I,J,K,N;
    APPAY T[1:12],U[1:12],TPREV[1:12],UPREV[1:12];

    ENDPTS ← GRAND(A);
    ENDPTS ← 0.5*(GRAND(B)+ENDPTS);
    SUMT ← 0.0;
    I ← N ← 1;
    H ← B-A;
ESTIMATE:
    T[1] ← H*(ENDPTS+SUMT);
    SUMU ← 0.0;

    X ← A-H/2.0;
    FOR J ← 1 STEP 1 UNTIL N DO
    BEGIN
      X ← X+H;
      SUMU ← SUMU+GRAND(X);
    END;
    U[1] ← H*SUMU;
    K ← 1;
TEST:
    IF ABS(T[K]-U[K]) <= EPS THEN
    BEGIN
      HAVIE ← 0.5*(T[K]+U[K]);
      GO TO EXIT;
    END;

    IF K ≠ I THEN
    BEGIN
      D ← D ↑ (2*K);
      T[K+1] ← (D*T[K]-TPREV[K])/(D-1.0);
      TPREV[K+1] ← T[K];
      U[K+1] ← (D*U[K]-UPREV[K])/(D-1.0);
      UPREV[K] ← U[K];
      K ← K+1;
      IF K = M THEN
      BEGIN
        HAVIE ← MASK;
        GO TO EXIT;
      END;
      GO TO TEST;
    END;

    H ← H/2.0;
    SUMT ← SUMT + SUMU;
    TPREV[K] ← T[K];
    UPREV[K] ← U[K];
    I ← I+1;
    N ← 2*N;
    GO TO ESTIMATE;
 EXIT:
  END;  ! END OF HAAVIE INTEGRATOR.;


  REAL PPOCEDUPE EXP2(X);
    VALUE X;
    REAL X;
    EXP2 ← EXP(-X*X);

  A ← 0.0;
  B ← 1.0;
  EPS ← 0.00005;
  MASK ← 9.99;
  ANSWER ← HAVIE(A,B,EPS,SQRT,12);
  WPITE("[C]");  PPINT(ANSWER,4,10);  WRITE("[C]");
  EPS ← 0.000001;
  A ← 0.0;
  B ← 4.3;
  ANSWER ← HAVIE(A,B,EPS,EXP2,12);
  WRITE("[C]");  PRINT(ANSWER,4,10);  WRITE("[C]");
 END;  ! END OF MAIN PROGPAM.
```

ISING

```
BEGIN
  COMMENT THIS IS ALGOPITHM 355 OF THE CACM ALGORITHM SECTION.
    PUBLISHED IN CACM 12.10 (OCT 1969) P.562.
    OUTER BLOCK WITH I/O AND OTHEP STATEMENTS INTPODUCED
    AND VALUE PAPTS AND PEMAINDEP OPEPATOR ADDED BY A. LUNDE.
    CAPNEGIE-MELLON UNIVERSITY, JULY 1972.;

  INTEGEP APPAY SEQU[1:100];
  INTEGER MAX,ONES,SHIFTS,I,UPPEP,MAXM1,IM1;

  PROCEDURE ISING(N,X,T,S); VALUE N,X,T;
    INTEGER N,X,T; INTEGEP APPAY S;
  BEGIN
    INTEGER K;
    INTEGER APPAY L,M[1:T DIV 2+1];

    PROCEDUPE SOPT(L,M,Z); VALUE Z;
      INTEGEP ARRAY L,M; INTEGER Z;
    BEGIN
      INTEGEP P,I,J,ML,ZB;
      FOR ML ← 1 STEP 1 UNTIL N DO S[ML] ← Z;
      R ← I ← 1;   ZB ← 1-Z;
AA:   J ← R+L[I]-1;
      FOR ML ← P STEP 1 UNTIL J DO S[ML] ← ZB;
      IF I+1 <= K THEN
        BEGIN R ← J+M[I]+1;  I ← I+1;  GO TO AA END;
      GO TO EXIT;
      WPITE("[C]");
      FOP ML ← 1 STEP 1 UNTIL N DO
        BEGIN
          IF (ML PEM 2) = 0 THEN WRITE("[C]---");
          PPINT(S[ML],2,0);
        END;
EXIT:
    END SORT;

    PPOCEDUPE BISORT(L,M);  INTEGER ARRAY L,M;
    BEGIN
      SOPT(L,M,0);   SORT(M,L,1)
    END BISORT;

    PPOCEDUPE COMPOSE(X,K,L,P);  VALUE X,K; INTEGER X,K;
      INTEGER APRAY L;  PROCEDUPE P;
    BEGIN
      INTEGEP I,A;
      IF X < K THEN GO TO CC;
      L[1] ← X-K+1;
      FOP I←2 STEP 1 UNTIL K DO L[I] ← 1;
      P;
      IF K <= 1 THEN GO TO CC;
      A ← 1;
BB:   IF L[A] > 1 THEN
        BEGIN
          L[A] ← L[A]-1;   L[A+1] ← L[A+1]+1;   P;
          IF A ≠ K-1 THEN A ← A+1;
          GO TO BB
        END; COMMENT  L[A] > 1 LOOP;

      L[A] ← L[A+1];   L[A+1] ← 1;   A ← A-1;
      IF A >= 1 THEN GO TO BB;
CC:
    END COMPOSE;

    K ← T DIV 2+1;
    IF (T PEM 2) = 1 THEN
      BEGIN
        PPOCEDUPE P1; BISOPT(L,M);
        PPOCEDUPE P2; COMPOSE(N-X,K,M,P1);

        COMPOSE(X,K,L,P2)
      END
    ELSE
      BEGIN
        PPOCEDUPE P3;  SOPT(L,M,0);
        PROCEDUPE P4;  COMPOSE(N-X,K-1,M,P3);
        PPOCEDUPE P5;  SOPT(M,L,1);
        PROCEDUPE P6;  COMPOSE(N-X,K,M,P5);

        COMPOSE(X,K,L,P4);
        COMPOSE(X,K-1,L,P6)
      END;

  END ISING;
```

Listing of the short subject algorithms

```
  WRITE("[C]TYPE UPPER BOUND FOR MAX[C]*");
  READ(UPPER);   WRITE("[C]");
  FOR MAX ← 3 STEP 1 UNTIL UPPER DO
  BEGIN
    MAXM1 ← MAX-1;
    FOR ONES ← 1 STEP 1 UNTIL MAXM1 DO
    BEGIN
      IMI ← IMIN(ONES,MAX-ONES);
      FOR SHIFTS ← 1 STEP 1 UNTIL IMI DO
        ISING(MAX,ONES,SHIFTS,SEQU);
    END;
  END;

END MAINPROGRAM;
```

BASIC VERSION OF PERT

```
300   DIM I(300),L(300),M(300)
400   DIM E(300),F(300),X(300)
410  N1 = 32
420  GOSUB 700
430  N1 = 27
440  GOSUB 700
450  STOP
700  REM  SUBROUTINE WORK
800   I(1) = 1
900   L(1) = 2
1000  E(1) = 2.5
1100  I(2) = 1
1200  L(2) = 3
1300  E(2) = 1.8
1400  I(3) = 1
1500  L(3) = 4
1600  E(3) = 3.0
1700  I(4) = 1
1800  L(4) = 10
1900  E(4) = 18.4
2000  I(5) = 2
2100  L(5) = 5
2200  E(5) = 4.2
2300  I(6) = 2
2400  L(6) = 6
2500  E(6) = 3.8
2600  I(7) = 2
2700  L(7) = 7
2800  E(7) = 6.7
2900  I(8) = 3
3000  L(8) = 6
3100  E(8) = 1.1
3200  I(9) = 3
3300  L(9) = 7
3400  E(9) = 1.3
3500  I(10) = 4
3600  L(10) = 7
3700  E(10) = 0.2
3800  I(11) = 6
3900  L(11) = 5
4000  E(11) = 6.6
4100  I(12) = 6
4200  L(12) = 8
4300  E(12) = 2.2
4400  I(13) = 7
4500  L(13) = 8
4600  E(13) = 4.9
4700  I(14) = 7
4800  L(14) = 9
4900  E(14) = 3.2
5000  I(15) = 7
5100  L(15) = 10
5200  E(15) = 1.1
5300  I(16) = 5
5400  L(16) = 11
5500  E(16) = 6.0
5600  I(17) = 8
5700  L(17) = 11
5800  E(17) = 6.0
5900  I(18) = 9
6000  L(18) = 11
6100  E(18) = 8.1
6200  I(19) = 10
6300  L(19) = 11
6400  E(19) = 0.7
6500  I(20) = 5
6600  L(20) = 12
6700  E(20) = 4.8
6800  I(21) = 8
6900  L(21) = 12
7000  E(21) = 0.7
7100  I(22) = 8
7200  L(22) = 14
7300  E(22) = 6.4
7400  I(23) = 10
7500  L(23) = 14
7600  E(23) = 3.8
7700  I(24) = 12
7800  L(24) = 13
7900  E(24) = 0.2
8000  I(25) = 11
8100  L(25) = 13
8200  E(25) = 2.5
8300  I(26) = 11
8400  L(26) = 14
8500  E(26) = 0.9
8600  I(27) = 6
8700  L(27) = 13
8800  E(27) = 11.1
8900  I(28) = 7
9000  L(28) = 5
9100  E(28) = 6.0
9200  I(29) = 11
9300  L(29) = 12
```

```
9400  E(29) = 7.3
9500  I(30) = 9
9600  L(30) = 12
9700  E(30) = 3.8
9800  I(31) = 14
9900  L(31) = 13
10000 E(31) = 0.7
10100 I(32) = 4
10200 L(32) = 10
10300 E(32) = 12.6
10500 T1 = 0.0
10700 PEM CALL PERT(N1,I1,L,E1,T1,N2,L2,E2,X1)
10800 GOSUB 13300
10805  PEM CALL PUTOUT
10810  GOSUB 10910
10830  RETURN
10900
10910  REM SUBROUTINE PUTOUT
11000 PRINT N2," EVENTS"
11100 RETUPN
11200 FOR I2 = 1 TO N2 STEP 1
11300 IF 0.001 > ABS((F(I2)-X(I2))) THEN 11700
11400 PRINT M(I2),F(I2),X(I2)
11600 GO TO 11900
11700 PRINT M(I2),F(I2),X(I2)," CRITICAL"
11900 NEXT I2
11990  PETURN
11998 PEM
11999 PEM
12000 REM SCAN(I2,I4,L2)
12010 PEM
12200 IF I2 = 1 THEN 13000
12400 FOR K1 = I2-1 TO 1 STEP -1
12600 IF I4 = M(K1) THEN 12700
12650 GO TO 12900
12700 I4 = K1
12800 PETUPN
12900 NEXT K1
13000 M(I2) = I4
13100 I4 = I2
13200 I2 = I2+1
13290 PETURN
13298 PEM
13299 PEM
13300 PEM PERT(N1,I1,L,E1,T1,N2,L2,E2,X1)
13400 REM
13600 I2 = 1
13700 FOR N3 = 1 TO N1 STEP 1
13780 I4 = L(N3)
13790 GOSUB 12000
13800 PEM CALL SCAN(I2,L(N3),L2)
13810 L(N3) = I4
13880 I4 = I(N3)
13890 GOSUB 12000
13900 REM CALL SCAN(I2,I(N3),L2)
13910 I(N3) = I4
14000 NEXT N3
14100 N2 = I2-1
14200 I5 = 1
14300 A1 = T1
14500 PEM      WHILE TPUE DO
14800 K2 = N2
14900 FOR I3 = 1 TO N2 STEP 1
15000 X(I3) = A1
15050 NEXT I3
15200 PEM         DO <BODY> WHILE K2 >< 0
15500 FOR N3 = 1 TO N1 STEP 1
15600 IF 0 >= M(I(N3)) THEN 16900
15800 PEM              CASE I5 OF
16000 ON I5 GO TO 16200,16600
16200 X2 = ABS(X(I(N3)))+E(N3)
16300 IF ABS(X(L(N3))) >= X2 THEN 16900
16350 X(L(N3)) = -X2
16400 GO TO 16900
16600 X2 = ABS(X(I(N3)))-E(N3)
16700 IF X2 >= ABS(X(L(N3))) THEN 16900
16750 X(L(N3)) = -X2
16900 NEXT N3
17100 FOP I3 = 1 TO N2 STEP 1
17200 IF M(I3) >= 0 THEN 17800
17300 IF X(I3) >= 0.0 THEN 18300
17400 M(I3) = ABS(M(I3))
17500 K2 = K2+1
17600 X(I3) = ABS(X(I3))
17700 GO TO 18300
17800 IF 0.0 > X(I3) THEN 18200
17900 M(I3) = -M(I3)
18000 K2 = K2-1
18100 GO TO 18300
18200 X(I3) = ABS(X(I3))
18300 NEXT I3
18400 IF K2 = 0.0 THEN 18700
18450 GO TO 15200
18700 ON I5 GO TO 19000,20500
18900 PEM      CASE 1
19000 I5 = 2
19100 FOR N3 = 1 TO N1 STEP 1
19200 I6 = I(N3)
19300 I(N3) = L(N3)
19400 L(N3) = I6
19500 NEXT N3
19600 A1 = 0.0
19700 FOP I3 = 1 TO N2 STEP 1
19800 F(I3) = X(I3)
19900 M(I3) = ABS(M(I3))
20000 IF A1 >= X(I3) THEN 20100
20050 A1 = X(I3)
20100 NEXT I3
20200 GO TO 20900
20400 PEM      CASE 2
20500 FOP I3 = 1 TO N2 STEP 1
20600 M(I3) = ABS(M(I3))
20650 NEXT I3
20700 PETURN
20900 GO TO 14500
21000 END
```

Listing of the short subject algorithms

BLISS VERSION OF PERT

```
MODULE BLI119(STACK(1000)) =
BEGIN
MACPO AOS(X) = (IF (X) GEQ 0.0 THEN (X) ELSE FNEG (X))$;
MACRO IABS(X) = (IF (X) GEQ 0 THEN (X) ELSE -(X))$;

EXTEPNAL OUTMSG,DECOUT,FLOUT;
FOPWAPD PUTOUT;

OWN NEVNTS;
OWN INIT[300],LAST[300],LINK[300];
OWN ESTIME[300],EARLYS[300],LATEF[300];

STPUCTURE VECT[I] = (.VECT+.I-1)<0,36>;
MAP VECT INIT:LAST:LINK:ESTIME:EARLYS:LATEF;

FUNCTION PERT(NMAX,IBEG,JEND,TE,ST,EMAX,LNK,ES,ATT) =
BEGIN

  STPUCTUPE PAPVEC[I] = (@.PAPVEC+.I-1)<0,36>;
  MAP PAPVEC IBEG:JEND:LNK:TE:ES:ATT;

  LOCAL IEX,ISX,ITX,KX;
  LOCAL AXX,XXX;

  FUNCTION SCAN(TOBJ) =
  BEGIN
    IF .IEX NEQ 1 THEN
    BEGIN
      DECR KX FROM .IEX-1 TO 1 BY 1 DO
        IF @.TOBJ EQL .LNK[.KX] THEN
        BEGIN (.TOBJ)<0,36> + .KX; PETURN  END
    END;
    LNK[.IEX] + @.TOBJ;  (.TOBJ)<0,36> + .IEX;  IEX + .IEX+1;
  END;   !SCAN;

  IEX + 1;
  INCP NX FPOM 1 TO .NMAX BY 1 DO
    BEGIN   SCAN(JEND[.NX]<0,0>);   SCAN(IBEG[.NX]<0,0>);   END;
  (.EMAX)<0,36> + .IEX-1;  ISX + 0;  AXX + .ST;

  WHILE 1 DO          ! WHILE TPUE DO
  ( KX + @.EMAX;
    INCR IEX2 FPOM 1 TO @.EMAX BY 1 DO ATT[.IEX2] + .AXX;

    DO        ! DO <BODY> WHILE .KX NEQ 0.
    ( INCR NX FROM 1 TO .NMAX BY 1 DO
      BEGIN
        IF .LNK[.IBEG[.NX]] GTR 0 THEN
        BEGIN
          CASE .ISX OF
          SET

            ! CASE 1
            BEGIN
              XXX + AOS(.ATT[.IBEG[.NX]]) FADR .TE[.NX];
              IF .XXX GTR ABS(.ATT[.JEND[.NX]])
                THEN ATT[.JEND[.NX]] + FNEG(.XXX);
            END;

            ! CASE 2
            BEGIN
              XXX + ABS(.ATT[.IBEG[.NX]]) FSBR .TE[.NX];
              IF .XXX LSS ABS(.ATT[.JEND[.NX]])
                THEN ATT[.JEND[.NX]] + FNEG(.XXX);
            END;
          TES;
        END;
      END;

      INCR IEX2 FROM 1 TO @.EMAX BY 1 DO
      BEGIN
        IF .LNK[.IEX2] LSS 0 THEN
        BEGIN
          IF .ATT[.IEX2] LSS 0 THEN
          BEGIN
            LNK[.IEX2] + IABS(.LNK[.IEX2]);   KX + .KX+1;
            ATT[.IEX2] + AOS(.ATT[.IEX2]);
          END;
        END ELSE
        IF .ATT[.IEX2] GEQ 0 THEN
        BEGIN  LNK[.IEX2] + -.LNK[.IEX2];  KX + .KX-1;  END
        ELSE ATT[.IEX2] + ABS(.ATT[.IEX2]);
      END;
    ) WHILE .KX NEQ 0;

    CASE .ISX OF
    SET

      ! CASE 1
            BEGIN
        ISX + 1;
        INCP NX FROM 1 TO .NMAX BY 1 DO
        BEGIN
          ITX + .IBEG[.NX];   IBEG[.NX] + .JEND[.NX];
          JEND[.NX] + .ITX;
```

```
          END;
          AXX + 0;
          INCR IEX2 FROM 1 TO @.EMAX BY 1 DO
          BEGIN
            ES[.IEX2] + .ATT[.IEX2];
            LNK[.IEX2] + IABS(.LNK[.IEX2]);
            IF .ATT[.IEX2] GTR .AXX THEN AXX + .ATT[.IEX2];
          END;
        END;   ! OF CASE 1

        ! CASE 2
        BEGIN
          INCR IEX2 FPOM 1 TO @.EMAX BY 1 DO
            LNK[.IEX2] + IABS(.LNK[.IEX2]);
          PETURN
        END;   ! OF CASE 2
      TES;
    )       ! END OF WHILE TPUE DO LOOP.
END;    ! PEPT;

FUNCTION WOPK(NACTS) =
BEGIN
  LOCAL TSTART;

   INIT[1] + 1;    LAST[1] + 2;    ESTIME[1] + 2.5;
   INIT[2] + 1;    LAST[2] + 3;    ESTIME[2] + 1.8;
   INIT[3] + 1;    LAST[3] + 4;    LSTIME[3] + 3.0;
   INIT[4] + 1;    LAST[4] + 10;   ESTIME[4] + 18.4;
   INIT[5] + 2;    LAST[5] + 5;    ESTIME[5] + 4.2;
   INIT[6] + 2;    LAST[6] + 6;    ESTIME[6] + 3.8;
   INIT[7] + 2;    LAST[7] + 7;    ESTIME[7] + 6.7;
   INIT[8] + 3;    LAST[8] + 6;    ESTIME[8] + 1.1;
   INIT[9] + 3;    LAST[9] + 7;    ESTIME[9] + 1.3;
   INIT[10] + 4;   LAST[10] + 7;   ESTIME[10] + 0.2;
   INIT[11] + 6;   LAST[11] + 5;   ESTIME[11] + 6.6;
   INIT[12] + 6;   LAST[12] + 8;   ESTIME[12] + 2.2;
   INIT[13] + 7;   LAST[13] + 8;   ESTIME[13] + 4.9;
   INIT[14] + 7;   LAST[14] + 9;   ESTIME[14] + 3.2;
   INIT[15] + 7;   LAST[15] + 10;   LSTIME[15] + 1.1;
   INIT[16] + 5;   LAST[16] + 11;   ESTIME[16] + 6.0;
   INIT[17] + 8;   LAST[17] + 11;   ESTIME[17] + 6.0;
   INIT[18] + 9;   LAST[18] + 11;   ESTIME[18] + 8.1;
   INIT[19] + 10;   LAST[19] + 11;   ESTIME[19] + 0.7;
   INIT[20] + 5;   LAST[20] + 12;   ESTIME[20] + 4.8;
   INIT[21] + 8;   LAST[21] + 12;   ESTIME[21] + 0.7;
   INIT[22] + 8;   LAST[22] + 14;   ESTIME[22] + 6.4;
   INIT[23] + 10;  LAST[23] + 14;   ESTIME[23] + 3.8;
   INIT[24] + 12;   LAST[24] + 13;   ESTIME[24] + 0.2;
   INIT[25] + 11;   LAST[25] + 13;   ESTIME[25] + 2.5;
   INIT[26] + 11;   LAST[26] + 14;   ESTIME[26] + 0.9;
   INIT[27] + 6;   LAST[27] + 13;   ESTIME[27] + 11.1;
   INIT[28] + 7;   LAST[28] + 5;   ESTIME[28] + 6.0;
   INIT[29] + 11;   LAST[29] + 12;   ESTIME[29] + 7.3;
   INIT[30] + 9;   LAST[30] + 12;   ESTIME[30] + 3.8;
   INIT[31] + 14;   LAST[31] + 13;   ESTIME[31] + 8.7;
   INIT[32] + 4;   LAST[32] + 10;   ESTIME[32] + 12.6;

   TSTAPT + 0.0;

   PEPT(.NACTS,INIT<0,0>,LAST<0,0>,ESTIME<0,0>,.TSTAPT,
        NEVNTS<0,0>,LINK<0,0>,EAPLYS<0,0>,LATEF<0,0>);
   PUTOUT(.NEVNTS,LINK<0,0>,EAPLYS<0,0>,LATEF<0,0>);

 END;  !  POUTINE WORK

 POUTINE PUTOUT(NEV,LK,EAS,XLF) =
 BEGIN
   STPUCTUPE PAPVEC[I] = (@.PAPVEC+.I-1)<0,36>;
   MAP PAPVEC LK:EAS:XLF;

   OUTMSG(0,PLIT '?M?J');  DECOUT(0,4,.NEV);
   OUTMSG(0,PLIT '  EVENTS?M?J');
   PETURN;
   INCR IX FROM 1 TO .NEV BY 1 DO
   BEGIN
     OUTMSG(0,PLIT '?M?J');  DECOUT(0,4,.LK[.IX]);
     FLOUT(0,.EAS[.IX],10,4);  FLOUT(0,.XLF[.IX],10,4);
     IF ABS(.EAS[.IX] FSBR .XLF[.IX]) LSS 0.001
       THEN OUTMSG(0,PLIT '  CPITICAL');
   END;
   OUTMSG(0,PLIT '?M?J');
 END;  ! POUTINE PUTOUT

 WOPK(32);
 WOPK(27);

 END
 ELUDOM
```

Listing of the short subject algorithms

FORTRAN VERSION OF PERT

```
      CALL WORK(32)
      CALL WORK(27)
      END
      SUBROUTINE WORK(NACTS)
C     INITIALIZE DATA AND CALL THE PROPER STUFF.
      DIMENSION INIT(300),LAST(300),LINK(300)
      DIMENSION ESTIME(300),EARLYS(300),XLATEF(300)

      INIT(1) = 1
      LAST(1) = 2
      ESTIME(1) = 2.5
      INIT(2) = 1
      LAST(2) = 3
      ESTIME(2) = 1.8
      INIT(3) = 1
      LAST(3) = 4
      ESTIME(3) = 3.0
      INIT(4) = 1
      LAST(4) = 10
      ESTIME(4) = 18.4
      INIT(5) = 2
      LAST(5) = 5
      ESTIME(5) = 4.2
      INIT(6) = 2
      LAST(6) = 6
      ESTIME(6) = 3.8
      INIT(7) = 2
      LAST(7) = 7
      ESTIME(7) = 6.7
      INIT(8) = 3
      LAST(8) = 6
      ESTIME(8) = 1.1
      INIT(9) = 3
      LAST(9) = 7
      ESTIME(9) = 1.3
      INIT(10) = 4
      LAST(10) = 7
      ESTIME(10) = 0.2
      INIT(11) = 6
      LAST(11) = 5
      ESTIME(11) = 6.6
      INIT(12) = 6
      LAST(12) = 8
      ESTIME(12) = 2.2
      INIT(13) = 7
      LAST(13) = 8
      ESTIME(13) = 4.9
      INIT(14) = 7
      LAST(14) = 9
      ESTIME(14) = 3.2
      INIT(15) = 7
      LAST(15) = 10
      ESTIME(15) = 1.1
      INIT(16) = 5
      LAST(16) = 11
      ESTIME(16) = 6.0
      INIT(17) = 8
      LAST(17) = 11
      ESTIME(17) = 6.0
      INIT(18) = 9
      LAST(18) = 11
      ESTIME(18) = 8.1
      INIT(19) = 10
      LAST(19) = 11
      ESTIME(19) = 0.7
      INIT(20) = 5
      LAST(20) = 12
      ESTIME(20) = 4.8
      INIT(21) = 8
      LAST(21) = 12
      ESTIME(21) = 0.7
      INIT(22) = 8
      LAST(22) = 14
      ESTIME(22) = 6.4
      INIT(23) = 10
      LAST(23) = 14
      ESTIME(23) = 3.8
      INIT(24) = 12
      LAST(24) = 13
      ESTIME(24) = 0.2
      INIT(25) = 11
      LAST(25) = 13
      ESTIME(25) = 2.5
      INIT(26) = 11
      LAST(26) = 14
      ESTIME(26) = 0.9
      INIT(27) = 6
      LAST(27) = 13
      ESTIME(27) = 11.1
      INIT(28) = 7
      LAST(28) = 5
      ESTIME(28) = 6.0
      INIT(29) = 11
      LAST(29) = 12
      ESTIME(29) = 7.3
      INIT(30) = 9
      LAST(30) = 12
      ESTIME(30) = 3.8
      INIT(31) = 14
      LAST(31) = 13
      ESTIME(31) = 0.7
      INIT(32) = 4
      LAST(32) = 10
      ESTIME(32) = 12.6

      TSTART = 0.0
      CALL PERT(NACTS,INIT,LAST,ESTIME,TSTART,
     1          NEVNTS,LINK,EARLYS,XLATEF)
      CALL PUTOUT(NEVNTS,LINK,EARLYS,XLATEF)
      RETURN

      END
      SUBROUTINE PUTOUT(NEV,LK,EAS,XLF)
      DIMENSION LK(1),EAS(1),XLF(1)

      TYPE 1000,NEV
1000  FORMAT (1X,14,7H EVENTS)
      RETURN
      DO 1 IX = 1,NEV,1
        IF (ABS((EAS(IX)-XLF(IX))) .LT. 0.001) GO TO 2
        TYPE 1001,LK(IX),EAS(IX),XLF(IX)
1001    FORMAT (1X,14,2F14.4)
        GO TO 1
2       TYPE 1002,LK(IX),EAS(IX),XLF(IX)
1002    FORMAT (1X,14,2F14.4,9H CRITICAL)
1     CONTINUE
      RETURN
      END
      SUBROUTINE SCAN(IEX,ITOBJ,LNK)
      DIMENSION LNK(1)
      IF (IEX .EQ. 1) GO TO 1
      LUCY = IEX-1
      DO 2 KX2 = 1,LUCY,1
        KX = LUCY-KX2+1
        IF (ITOBJ .NE. LNK(KX)) GO TO 2
        ITOBJ = KX
        RETURN
 2    CONTINUE
 1    LNK(IEX) = ITOBJ
      ITOBJ = IEX
      IEX = IEX+1
      END
      SUBROUTINE PERT(NMAX,IBEG,JEND,TE,ST,MAXE,LNK,ES,AT)
      DIMENSION IBEG(1),JEND(1),LNK(1),TE(1),ES(1),AT(1)

      IEX = 1
      DO 1 NX = 1,NMAX,1
        CALL SCAN(IEX,JEND(NX),LNK)
        CALL SCAN(IEX,IBEG(NX),LNK)
 1    CONTINUE
      MAXE = IEX-1
      ISX = 1
      AXX = ST

C     WHILE TRUE DO

 2    CONTINUE
        KX = MAXE
        DO 3 IEX2 = 1,MAXE,1
 3      AT(IEX2) = AXX

C       DO <BODY> WHILE KX .NE. 0

 6      CONTINUE
          DO 4 NX = 1,NMAX,1
            IF (LNK(IBEG(NX)) .LE. 0) GO TO 4

C           CASE ISX OF

            GO TO (101,102),ISX

 101        XXX = ABS(AT(IBEG(NX)))+TE(NX)
            IF (XXX .GT. ABS(AT(JEND(NX)))) AT(JEND(NX)) = -XXX
            GO TO 4

 102        XXX = ABS(AT(IBEG(NX)))-TE(NX)
            IF (XXX .LT. ABS(AT(JEND(NX)))) AT(JEND(NX)) = -XXX

 4        CONTINUE

          DO 7 IEX2 = 1,MAXE,1
            IF (LNK(IEX2) .GE. 0) GO TO 8
            IF (AT(IEX2) .GE. 0.0) GO TO 7
            LNK(IEX2) = IABS(LNK(IEX2))
            KX = KX+1
            AT(IEX2) = ABS(AT(IEX2))
            GO TO 7
 8          IF (AT(IEX2) .LT. 0.0) GO TO 9
            LNK(IEX2) = -LNK(IEX2)
            KX = KX-1
            GO TO 7
 9          AT(IEX2) = ABS(AT(IEX2))
 7        CONTINUE
```

```
      IF (KX .NE. 0) GO TO 6

C     END OF DO <BODY> WHILE KX = 0.

      GO TO (201,202),ISX

C     CASE 1
 201  ISX = 2
      DO 12 NX = 1,NMAX,1
        ITX = IBEG(NX)
        IBEG(NX) = JEND(NX)
        JEND(NX) = ITX
 12   CONTINUE
      AXX = 0.0
      DO 11 IEX2 = 1,MAXE,1
        ES(IEX2) = AT(IEX2)
        LNK(IEX2) = IABS(LNK(IEX2))
        IF (AT(IEX2) .GT. AXX) AXX = AT(IEX2)
 11   CONTINUE
      GO TO 200

C     CASE 2
 202  DO 10 IEX2 = 1,MAXE,1
 10   LNK(IEX2) = IABS(LNK(IEX2))
      RETURN
 200  CONTINUE
     GO TO 2
C    END OF WHILE TRUE DO LOOP.
     END
```

```
THE 5 VERSIONS OF AITKEN, ALL IN ONE PROGRAM.
VERSION SELECTED BY CASE INDEX.

MODULE INTERPOL(STACK,TIMER=EXTERNAL(SIX12)) =
BEGIN

GLOBAL                    ! VARIABLES INITIALIZED BY DDT
  MAXC,                   ! UPPER LIMIT FOR LOOP
  HSTEP,                  ! STEP LENGTH DURING INTERPOLATION LOOP
  TRACECASE;              ! SELECTS ROUTINE TO BE TRACED

BIND
  NMAX = 10,              ! MAXIMAL NUMBER OF POINTS.
  TABSIZ = 201;           ! SIZE OF FUNCTION TABLE.

OWN
  X,
  ABSCIS[TABSIZ],         !ABSCISSAE OF FUNCTION TABLE
  ORDIN[TABSIZ];          !FUNCTION VALUES.

EXTERNAL
  LOG;

%) - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                 VERSION A
 - - - - - - - - - - - - - - - - - - - - - - - - - - - -(%
ROUTINE AAITKEN(XT,YT,XX,N,L) =
BEGIN
  REGISTER HI, LO, I;

  OWN
    X[NMAX],      ! ABSCISSAE
    DX[NMAX],     ! ABSCISSAE DIFFERENCE.
    Y[NMAX],      ! OLD FUNCTION VALUES.
    Z[NMAX];      ! NEW FUNCTION VALUES.

  STRUCTURE PARVEC[I] = (@.PARVEC+.I)<0,36>;
  MAP PARVEC XT:YT;

  IF .XX EQL .XT[.L] THEN RETURN .YT[.L];

  ! PREPARE AND PERFORM BINARY SEARCH FOR RIGHT INTERVAL.

  LO ← 0;   HI ← .L;  I ← .L/2;

  WHILE (.HI-.LO) GTR 1 DO
  ( ! LOOP INVARIANT IS:
    ! .XT[.LO] LEQ .XX LSS .XT[.HI]

    IF .XX EQL .XT[.LO] THEN RETURN .YT[.LO];
    IF .XX LSS .XT[.I] THEN HI ← .I ELSE LO ← .I;
    I ← (.HI+.LO)/2
  );
    ! NOW .XT[.LO] LEQ .XX LSS .XT[.LO+1]

  IF (LO ← .LO-.N/2+1) LSS 0 THEN LO ← 0;
  IF .LO + .N - 1 GTR .L THEN LO ← .L-.N+1;

  ! NOW READY TO INTERPOLATE,
  ! USING POINTS .LO, .LO+1,.....LO+.N-1.
  ! FIRST INITIALIZE LOCAL TABLE.

  LO ← .LO-1;
  INCR J FROM 0 TO .N-1 DO
  ( X[.J] ← .XT[LO ← .LO+1];
    Y[.J] ← .YT[.LO];
    DX[.J] ← .XT[.LO] FSBR .XX;
   !OUTINT(.J);!OUTINT(.LO);!OUTFL9(.X[.J]);
   !OUTFL9(.Y[.J]); !OUTFL9(.DX[.J]); !CRLF();
  );

  ! NOW COMPUTE SUCCESSIVE APPROXIMATIONS
  ! USING SUCCESSIVELY MORE POINTS

  INCR J FROM 0 TO .N-2 DO
  ( INCR K FROM .J+1 TO .N-1 DO
    ( Z[.K] ← ((.Y[.J] FMPR .DX[.K]) FSBR (.Y[.K] FMPR .DX[.J]))
                  FDVR (.X[.K] FSBR .X[.J]);
     !OUTFL9(.Z[.K]); !CRLF();
    );
    INCR K FROM .J+1 TO .N-1 DO Y[.K] ← .Z[.K]
  );

  ! NOW READY TO DELIVER VALUE:

  .Z[.N-1]
 END;   ! ROUTINE AAITKEN.
```

Listing of the short subject algorithms

```
%> - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                  VERSION L
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<%
ROUTINE INDEX(XTAB,L,N,X) =
BEGIN
%>
  FIND THE INDEX OF THE ELEMENT IN XTAB WHICH IS THE FIRST
  OF THE N ELEMENTS CLOSEST TO X
<%

STRUCTURE IVEC[I] = (@.IVEC+.I)<0,36>;
MAP IVEC XTAB;

LOCAL K,S,T;

! FIND K S.T. .XTAB[.K] LEQ .XTAB[.K+1]

INCR I FROM 1 TO .L DO
( IF .X EQL .XTAB[.I] THEN (K ← .I; EXITLOOP);
  IF .X LEQ .XTAB[.I] THEN (K ← .I-1; EXITLOOP);
);

! FIND START AND FINISH ELEMENTS DISREGARDING XTAB ARRAY BOUNDS.

  S ← .K-.N/2+1;    T ← .K+.N/2;
  IF (.N MOD 2) EQL 1 THEN
    IF (.X FSBR .XTAB[.K]) LSS (.XTAB[.K+1] FSBR .X)
        THEN S ← .S-1  ELSE T ← .T+1;

! ADJUST START ELEMENT TO CONFORM TO ARRAY BOUNDS.

  IF .S LSS 0 THEN S ← 0 ELSE IF .T GTR .L THEN S ← .S-.T+.L;

  RETURN .S

END;   ! ROUTINE INDEX.

ROUTINE LAITKEN(XTAB,YTAB,X,N,L) =
BEGIN
! N POINT INTERPOLATION.

STRUCTURE IVEC[I] = (@.IVEC+.I)<0,36>;
STRUCTURE MATRIX[I,J] = [I*J] (.MATRIX+.J+J*.I)<0,36>;

MACRO DET(A,B,C,D) = ((A FMPR D) FSBR (B FMPR C))$;

OWN MATRIX IN[10,10];
OWN XC[10];
LOCAL J;
MAP IVEC XTAB:YTAB;

J ← INDEX(.XTAB,.L,.N,.X);

! INITIALIZE XC[0:.N-1] TO .XTAB[.J:.J+.N-1]
INCR I FROM 0 TO .N-1 DO XC[.I] ← .XTAB[.I+.J];

! INITIALIZE IN[0:.N-1,0] TO YTAB[.J:.J+.N-1]
INCR I FROM 0 TO .N-1 DO IN[.I,0] ← .YTAB[.I+.J];

! GO
INCR J FROM 1 TO .N-1 DO
  INCR K FROM 1 TO .J DO
    IN[.J,.K] ← DET((.IN[.K-1,.K-1]),(.XC[.K-1] FSBR .X),
                 (.IN[.J,.K-1]),(.XC[.J] FSBR .X))
              FOVR (.XC[.J] FSBR .XC[.K-1]);

RETURN .IN[.N-1,.N-1]
END;  ! ROUTINE LAITKEN.
```

```
%> - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                  VERSION G
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<%
FUNCTION GAITKEN(XTAB,YTAB,X,N,L) =
BEGIN LOCAL XX:YY[10],LB ; BIND N1=.N-1 ;
    ! XX WILL HOLD X(I)-X FOR THE DATA POINTS CHOSEN, AND
    ! YY THE INTERPOLATED VALUES.
      BIND XT=.XTAB , YT=.YTAB ; MAP XT,YT ;
      LB←( LOCAL I,K ;
          I←0 ;
          WHILE .X GTR .XT[.I] AND .I LSS .L DO I←.I+1 ;
            ! I NOW HOLDS THE INDEX OF THE FIRST X(..)
            ! THAT IS GEQ X.
          K←.I-.N/2 ;
          IF .K LSS 0 THEN 0
                  ELSE IF .K GTR .L-.N+1 THEN .L-.N+1
                                        ELSE .K         );
           ! LB NOW HOLDS THE INDEX OF OUR SMALLEST BASE POINT.
           ! INITIALIZE XX AND YY.
      INCR I FROM 0 TO N1 DO
          ( XX[.I]←.XT[.LB+.I] FSBR .X ;
            YY[.I]←.YT[.LB+.I]          );
          ! INTERPOLATION EXACTLY ACCORDING TO
          ! SCHEME OF GIVEN REFERENCE.
          ! EACH I-ITERATION GIVES VALUES OF I-TH DEGREE.
      INCR I FROM 1 TO N1 DO
          (MACRO II=.I-1$ ;
          INCR J FROM .I TO N1 DO
               YY[.J]←    (      .YY[II] FMPR .XX[.J]
                             FSBR .YY[.J] FMPR .XX[II] )
                        FOVR (.XX[.J] FSBR .XX[II])      );
      .YY[N1]
 END;  ! GAITKEN
```

```
%> - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                  VERSION B
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<%
ROUTINE BAITKEN(XTAB,YTAB,X,N,L) =
BEGIN

STRUCTURE IVEC[I] = (@.IVEC+.I)<0,36>;
MAP IVEC XTAB:YTAB;
OWN VECTOR C[10],XX[10];
REGISTER B,E;

B ← XTAB[0];    E ← XTAB[.L-1];
WHILE (.E-.B) GTR 1 DO
  IF @((.B+.E)/2) GTR .X THEN E ← (.B+.E)/2 ELSE B ← (.B+.E)/2;

IF (B ← .B-.N/2+1) LSS XTAB[0] THEN B ← XTAB[0]
  ELSE IF .B GTR XTAB[.L-.N+1] THEN B ← XTAB[.L-.N+1];
E ← YTAB[.B-XTAB[0]];

DECR I FROM .N-1 TO 0 DO
( XX[.I] ← @.B;  C[.I] ← @.E;  B ← .B+1;  E ← .E+1);

DECR I FROM .N-1 TO 1 DO
  DECR J FROM .I-1 TO 0 DO
    C[.J] ← ((.C[.I] FMPR (.XX[.J] FSBR .X)) FSBR
             (.C[.J] FMPR (.XX[.I] FSBR .X))) FOVR
              (.XX[.J] FSBR .XX[.I]);

 .C[0]
END;   ! ROUTINE BAITKEN
```

```
%> - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                   VERSION E
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -<%
ROUTINE EAITKEN(XTAB,YTAB,XP,N,L) =
BEGIN

OWN VECTOR C[10],XX[10],XXX[10];
BEGIN ! THIS BLOCK SAVES ONE INSTR. IN THE ENTRY CODE AND ONE
      ! IN THE EXIT CODE SINCE WE NOW ONLY USE 4 PEGISTERS.
REGISTER B,E,X;

B ← .XTAB;   E ← .XTAB+2+.L-.N;  X ← .XP;
WHILE .E GTR (.B+1) DO
  IF @((.B+.E)/2) GTR .X THEN E ← (.B+.E)/2 ELSE B ← (.B+.E)/2;

IF (B ← .B-.N/2+1) LSS .XTAB THEN B ← .XTAB;
E ← .YTAB+.B-.XTAB;

DECR I FROM .N-1 TO 0 DO
( XXX[.I] ← (XX[.I] ← @.B) FSBR .X;  C[.I] ← @.E;
  B ← .B+1;  E ← .E+1
);
END;   ! OF THE BLOCK THAT SAVES US ENTRY/EXIT CODE.

DECR I FROM .N-1 TO 1 DO
  DECR J FROM .I-1 TO 0 DO]
    C[.J] ← ((.C[.I] FMPR .XXX[.J]) FSBR
            (.C[.J] FMPR .XXX[.I])) FOVR
              (.XX[.J] FSBR .XX[.I]);

 .C[0]
END;   ! ROUTINE EAITKEN




ROUTINE TEST(IPO,H0) =
BEGIN

LOCAL
  J,
  H, HMAX, HMIN,
  X,
  Y,
  DY,
  FACT;


H ← .H0;   FACT ← 1.05;   X ← 1.0;
HMAX ← .H0 FMPR 3.0;   HMIN ← .H0 FMPR 0.2;

INCR I FROM 0 TO TABSIZ-1 BY 1 DO
( ABSCIS[.I] ← .X;
  IF .I GTR 0 THEN ( IF .ABSCIS[.I] LEQ .ABSCIS[.I-1] THEN );
  ORDIN[.I] ← LOG(.X);
  X ← .X FADR .H;   H ← .H FMPR .FACT;
  IF .H GTR .HMAX THEN (X ← .X FADR .H FDVR 3.0;
                        FACT ← 0.95);
  IF .H LSS .HMIN THEN FACT ← 1.05;
);

INCR COUNT FROM 1 TO .MAXC DO
( X ← 1.0;   H ← .HSTEP;
  WHILE .X LEQ .ABSCIS[TABSIZ-1] DO
  ( INCR I FROM 2 TO NMAX DO
       (.IPO)(ABSCIS<0,0>,ORDIN<0,0>,.X,.I,TABSIZ-1);
    X ← .X FADR .H
  );
); ! END OF TIMING LOOP
END;  ! OF ROUTINE TEST.

CASE .TRACECASE OF
SET
%0%  TEST(AAITKEN<0,0>,0.1);
%1%  TEST(LAITKEN<0,0>,0.1);
%2%  TEST(GAITKEN<0,0>,0.1);
%3%  TEST(BAITKEN<0,0>,0.1);
%4%  TEST(EAITKEN<0,0>,0.1);
TES;

END
ELUDOM
```