SLAC-205
STAN-CS-78-658

SLAC-205
STAN-CS-78-658

Analysis and Performance of Computer Instruction Sets

Leonard Jay Shustek

SLAC Report No. 205

STAN-CS-78-658

January 1978

Under Contract with the

Department of Energy

Contract No. EY-76-C-03-0515

STANFORD LINEAR ACCELERATOR CENTER

Stanford University · Stanford, California

REPRODUCED BY
U.S. DEPARTMENT OF COMMERCE
NATIONAL TECHNICAL
INFORMATION SERVICE
SPRINGFIELD, VA 22161

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights.

SLAC REPORT 205
STAN-CS-78-658
UC 32

ANALYSIS AND PERFORMANCE OF COMPUTER INSTRUCTION SETS*

LEONARD JAY SHUSTEK

STANFORD LINEAR ACCELERATOR CENTER

STANFORD UNIVERSITY

Stanford, California 94305

PREPARED FOR THE DEPARTMENT OF ENERGY

UNDER CONTRACT NO. EY-76-C-03-0515

January 1978

Printed in the United States of America. Available from National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22161. Price: Printed Copy $9.00; Microfiche $3.00.

*Ph.D. Dissertation

ABSTRACT

This study is concerned with measurements of the characteristics of current use of digital computer instruction sets, and an analysis of those measurements to produce conclusions useful for future design evaluation. The results include conclusions about computer architectures and their implementations, as well as techniques generally applicable to studies of this type.

In order to apportion the time spent by an executing program among the various system components, a model of high-performance computers is derived from instruction timing formulas, with compensation for pipeline and cache memory effects. The model is used to predict the performance of the IBM 370/168 and the Amdahl 470 V/6 on specific programs, and the results are verified by comparison with actual performance. This technique is one which is not as difficult as full hardware simulation, yet yields information about specific implementations which can be used to detect bottlenecks in processor execution and to predict the effect of changes to the processor.

The data for the performance analysis is obtained by interpretively tracing a variety of benchmark programs for the 370 architecture. In addition to the data needed for the performance evaluation, these traces are analyzed to provide information about the characteristics and use of the instruction set itself. Some of the issues discussed are: opcode distributions, instruction length, branch analysis, branch and execution distance, opcode pair distributions, displacement values, address register utilization, operand lengths, cache memory effects, and pipeline breaks.

Some of the same measurements are made and discussed for other instruction set architectures, including a current microprocessor (the INTEL 8080), a popular minicomputer (the DEC PDP11) and a pseudo-computer used as an intermediate step for PASCAL compilation or interpretation (PCODE).

Using the data and techniques developed, some predictive models are investigated. As an example of the effect of refinement of an implementation, an improved mechanism for conditional branch prediction is proposed and simulated. The performance of a radically different implementation of the 370 is examined and its performance is predicted. Finally, some of the data gathered is used as the basis for a model of the instruction fetch process; that model is solved analytically.

The last chapter summarizes the architectural conclusions and recommendations which result from this study. The emphasis is on design rules which should be followed to allow instruction sets to be implemented with high-performance processors.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Forest Baskett, for his inspiration, advice, and patience. I would also like to thank the other members of my committee -- John Wakerly and Gio Wiederhold -- for their careful and constructive reading of the thesis.

I especially want to express my gratitude to Bernard Peuto for being a critic, confidant, and friend when I needed it most.

Most of this work was done in the Computation Research Group of the Stanford Linear Accelerator Center, supported by the U.S. Department of Energy under contract E(Ø43)515.

TABLE OF CONTENTS

Acknowledgments iii

Chapter — page

I. INTRODUCTION 1

- Scope 1
- Motivation 3
- Approach 6
- Use of Interpretive Traces 7
- Organization of the Thesis 9

II. PREVIOUS WORK 12

- System-Level Performance Evaluation 13
- Benchmarking For Processor Evaluation 14
- Processor Modeling 16
- Measurement of Language Characteristics 17
- Measurement and Evaluation of Instruction Sets 18
- Instruction Timing Studies 22

III. INSTRUCTION TIMING MODEL FOR MACHINE EVALUATION 23

- The Model 24
  - The Technique 25
  - Evaluation of Important Factors 27
  - Instruction Timing Formulas 33
  - Measurement Approximations 36
  - Description of the Measurement System 39
- Verification 43
  - Cache Miss Penalty 43
  - SVC Times 46
  - Benchmark Jobs 47
  - Validation 49

IV. CHARACTERISTICS OF THE IBM 37Ø ARCHITECTURE 55

- Opcode Distributions 55
- Instruction Length 67
- Branch Analysis 68
- Branch and Execution Distances 71
- Opcode Pair Distribution 74
- Displacement values 78

Register Use 78
Operand Lengths 81
LM/STM 82
Character Instructions 83
Decimal Instructions 86
Memory Reference and Cache Effects 88
Pipeline Effects for the 470 89
Summary 90

V. OTHER ARCHITECTURE CHARACTERISTICS 92

INTEL 8080 92
Opcode Distribution 94
Opcode Pairs 95
Instruction Length 99
Branch and Execution Distance 100
Memory References 102
Summary 103
PASCAL P-Machine 104
Opcode Distribution 104
Opcode Pairs 105
Branch and Execution Distances 105
Summary 105
DEC PDP11 108
Opcode Distribution 108
Addressing Modes 109
Summary 115

VI. PREDICTIVE MODELS 118

370 Branch Improvement 118
Prediction of New Processor Performance 126
Markov Instruction-Fetch Model 130

VII. ARCHITECTURAL CONCLUSIONS 139

Time is of the Essence 140
Eliminate Decimal Instructions 141
String Moves are Dangerous 143
Watch Out for Unusual Special Cases 145
Indirect Addressing - NO! 147
Immediate Addressing - YES! 148
Beware of Expensive Software Conventions 148
Architectural Inconveniences Can Be Overcome by Compiler Optimization 149
The Importance of Branches 150
The Consequences of Small Execution Distances ... 151
Address Displacement Size Should Be Large 153
Are There Missing Instructions? 153
The Effect of Task Switches on Cache Contents ... 154
Epilogue 156

REFERENCES 157

Appendix page

A. INSTRUCTION MNEMONICS 163

IBM 370 Instructions 163
INTEL 8080 Instructions 165
DEC PDP11 Instructions 166
PASCAL PCODE Instructions 167

III

LIST OF TABLES

Table — page

1. SVC Times and Cache Requests 45
2. Program Characteristics 49
3. Predicted and Benchmark Times 53
4. Instruction lengths 68
5. Branch Instructions 69
6. Instructions Which Caused Branches, Sorted By Frequency 71
7. Execution Distances 73
8. Opcode Pairs 75
9. Register Use for RX Instructions 81
10. 8080 Dynamic Instruction Statistics 94
11. 8080 Dynamic Opcode Pairs 98
12. 8080 Static Opcode Pairs 100
13. Types of 8080 Jumps 101
14. 8080 Execution Distance 102
15. Pascal P-CODE Opcode Pairs 106
16. PDP11 Addressing Modes - All Operands (Static) 109
17. PDP11 Addressing Modes - By Operand (Static) 112
18. PDP11 Two-operand Addressing Modes 112
19. Special Uses of PDP11 MOV/MOVB 114
20. Special Uses of PDP11 2-Operand Instructions 115
21. Branch Prediction Success For The 168 120

22. Branch Success as a Function of Branch Direction 121
23. Conditional Branch Consistency and Prediction Success 123
24. Branch Prediction Success vs. Number of Executions 123
25. Measured vs Predicted Execution Distances 137

IV

LIST OF FIGURES

Figure ... page

1. Measurement System ... 42
2. Static/Dynamic Count, and Time by Opcode - FORTC ... 57
3. Static/Dynamic Count, and Time by Opcode - FORTGO ... 58
4. Static/Dynamic Count, and Time by Opcode - PL1C ... 59
5. Static/Dynamic Count, and Time by Opcode - PL1GO ... 60
6. Static/Dynamic Count, and Time by Opcode - COBOLC ... 61
7. Static/Dynamic Count, and Time by Opcode - COBOLGO ... 62
8. Static/Dynamic Count, and Time by Opcode - LINSY2 ... 63
9. Dynamic Opcode Counts - FFT programs ... 64
10. Dynamic Opcode Counts - Language Processors ... 65
11. Static/Dynamic Count, and Time by Opcode - LASS ... 66
12. Branch Distances ... 72
13. Execution Distance Between Branches ... 75
14. Distribution of Displacement Values ... 79
15. Number of Registers for LM/STM ... 83
16. MVC operand length ... 85
17. Intel 8080 Dynamic Instruction Count, and Time by Opcode ... 96
18. Intel 8080 Static Opcode Distribution ... 96

19. PDP11 and Pcode Static Opcode Distributions ... 106
20. Conditional Branch Successes ... 123
21. Predicted 168/E Performance Compared to IBM 168 ... 129
22. Instruction Fetch Model 1 ... 131
23. Modified Instruction Fetch Model 1 ... 132
24. Execution Distance Distributions ... 133
25. Instruction Fetch Model 2 ... 134
26. Modified Instruction Fetch Model 2 ... 136
27. Alternate MOVE schemes ... 144

Chapter I

INTRODUCTION

1.1 SCOPE

The goal of this study is to examine, understand, and justify some of the many architectural decisions that are involved in the design of a computer. We will concentrate on an analysis of instruction sets, and will, by measurement of the characteristics of their current use, produce conclusions that are useful for future design efforts.

By interpretively tracing programs on a number of existing architectures, a great deal of data is produced which can be used for such architectural studies. The results discussed here can be divided into six types:

1. Particular conclusions about the architecture and implementation of the computers studied,
2. General conclusions about machine architecture which can be inferred,
3. Proposals for improvement which are indicated by these conclusions,

4. A set of techniques and tools which are generally applicable to studies of this type,
5. The use of these techniques to predict the effect of proposed design changes, and
6. A collection of quantitative data which can be used for further study or as a basis of comparison for other architectures.

The techniques will be applied to the extremes of current computers: from high performance implementations of sophisticated processors (the IBM 370/168 and Amdahl 470 V/6), to a comparatively simple LSI microprocessor (the Intel 8080). Particularly for the larger machines, it will be important to demonstrate that the approximation involved in analysis can be verified using the real machines. The true value of these techniques, however, lies in the ability both to predict the effect of changes to particular implementations, and to draw conclusions about the instruction set architectures.

Although a range of architectures is examined, the emphasis is on the characteristics of instruction sets for high performance implementations. As a specific example, the IBM 370 is studied in detail to provide a framework for discussing the conflicting goals of high speed and instruction set complexity.

1.2 MOTIVATION

> "Errors using even inadequate data are much less than those using no data at all."
>
> -- Charles Babbage (1792-1871)

The design of a modern computer is a complex process that must first begin with an understanding of the general goals, the available resources, and the technological limitations. Even within these constraints, however, the space of possible designs for even modest subparts of the system is extremely large, and hundreds of major decisions must be made before the specification is complete.

This lengthy design process requires that a large number of difficult architectural questions be resolved rather early in the development of the machine. Some of the decisions will have only a small effect on the ultimate performance and others will be crucial, but is it often difficult to distinguish between the two before the development is complete. There are problems of architectural definition which apply to all members of a computer "family" as well as more specific questions that arise during implementation of particular models.

Unfortunately, there is very little information available which can serve as a guide for any of these issues. There is almost no formal theory of computer architecture which is of use to the designer, and there is a

notable lack of retrospective analyses of existing computers in open literature. The result is that the success of the design often depends on the intuition, experience, and good luck of the designers.

The design of the instruction set is an area which can particularly benefit from measurement and analysis of existing computers, especially for architectures which will eventually have high performance implementations. It is important to avoid features for which the commonly occurring cases cannot be easily optimized, yet there are many such examples in popular architectures. In other cases, small changes to the specification of the instruction formats (start/end versus start/length for sequences, for example) can make a large difference in implementation cost yet are almost transparent even to assembly language programmers.

An equally important and perhaps even more fruitful application of such analysis is for the improvement of existing designs. The high cost of software development (and the continuing dependence on machine languge progamming) dicates that radical changes in machine architecture be made only at infrequent intervals. Although users of a given computer are often willing to expound at length on the deficiencies which seem to them to be the most obvious, it requires careful and precise measurements to identify those areas where efforts to improve the design

will have the greatest effect. It is tempting to concentrate on the defects which most offend aesthetic sensitivities rather than on those whose elimination will do the most to improve performance overall.

Because of the likelihood of later extensions, it is important that the initial design of an instruction set be flexible enough to adapt to such changes. This at the very least requires some unused slots in the encoding scheme, and may also require that some features be left out in order to avoid incompatiblity with likely future additions. The lack of hindsight is a problem, but even later measurements may not be enough to show what the correct decision should have been. It often is easy enough to recognize and measure the effects of mistakes in architecture or implementation, but it is much harder to see what the good features are. Failures are much more obvious than sucesses.

Although it is primarily the computer architecture that is being studied, it is crucial that any such measurements be based on performance while executing typical real programs. The characteristics of a computer system is as much a product of the software which is used as the design of the hardware, and any study which is based only on a theoretical examination of the properties of the design must bear the burden of demonstrating whether it applies to the way the computer is really used.

1.3 APPROACH

Any study concerned with the way computers are used must have models for both the computers and the programs which are to run on them. The difference between various models is most often a difference in the level of detail considered; each model makes assumptions about which parameters are significant and which can be neglected.

In queuing theory representations of computer systems, the programs are modeled by their resource request and use characteristics. At this level, the detailed sequence of operations performed by a program are either generated randomly based on known or measured averages (for simulation studies), or are represented by simple mathematical models (where analytic techniques are feasible). This approach has had great success for subsystem-level modeling, where the basic units are memories, processors, and I/O devices. This success is based primarily on the accuracy with which requests to resources can be modeled in a simple, independent fashion.

When the basic units to be studied are at a more detailed level -- registers, cache memory locations, execution pipeline slots -- the queuing model approach fails because no statistically simple model of the resource requests and service can be supplied. The way programs behave at this level is complex, time dependent, and varies

greatly from program to program. For this reason it is best to avoid constructing a model of program activity in favor of using program execution directly to supply the needed data. A convenient way to accomplish this is to use a software interpreter which simulates the execution of a real program exactly as the hardware would but collects, as a side effect, information about that program's use of hardware resources. It is that approach which has been used to collect much of the data used in this study.

1.4 USE OF INTERPRETIVE TRACES

A software interpreter has a number of attractive elements which contribute to its usefulness for instruction set analysis and performance evaluation. One of the most important aspects is that a real (non-synthetic) program is executed while the data are collected, so that the resulting evaluation concentrates precisely on the important resources that are used by that program. Put another way, the interpretive technique correctly weights the costs of basic operations with their true frequency of use. We will see later how this can be combined with design information to yield quantitative information that is both specific to particular implementations and applicable to the architecture generally[1].

---

[1] The term "implementation" will be used throughout to mean a particular computer design which executes a given instruction set; the design of the instruction set itself will be called the "architecture". Many of the instruction

In addition to hardware-oriented data, an interpreter also provides program-level information that is useful to programmers, compiler-writers, and operating-system builders. The results of particular compiler optimizations, or the cost of certain software conventions can be determined and evaluated. Equally important is the simple satisfaction of curiosity about how the computer is really being stressed. Although the individual instructions were written by the programmers, their aggregate effect (aside from occasionally producing correct results) often comes as a surprise.

An approach often used to gather very similar data is to write (or build!) a detailed simulator of the hardware for a particular architecture implementation. In comparison to this full simulation of the hardware at a detailed level, instruction-level interpretation is relatively easy to do. Although more implementation-specific information can be obtained from a hardware simulator, the high cost of writing and using it has discouraged its casual use. It is generally difficult to modify, expensive to run, and very specific to the particular implementation.

---

architectures we look at will have several implementations ("models") consituting a family of computers, such as the DEC PDP11 and IBM 370 series. An abstract processor which implements a particular architecture is often called an ISP ("Instruction Set Processor").

Rather than get information which is tightly coupled to the implementation and the hardware, we would like a similar technique which is closer to the architecture and program levels. An instruction interpreter with some knowledge of the implementation is a good compromise that provides a large part of the relevant data at moderate cost.

Much of the information used in this study comes from data produced by an instruction interpreter, augmented by detailed simulation of specific subsystems where necessary. This will be referred to as "dynamic" analysis, that is, analysis of executing programs as seen by the instruction processor, and provides most of the information of interest about performance and resource utilization. In addition, other information will come from "static" analysis of programs before execution; this provides insight into memory requirements (both size and bandwidth), and influences the choice of instruction encoding technques.

1.5 ORGANIZATION OF THE THESIS

The remaining chapters of the thesis are as follows:

Chapter 2 discusses some of the previous work on processor or architecture evaluation, and shows how the work here is related.

Chapter 3 describes the basic instruction timing model used for processor evaluation. The methodology is

explained, and the various factors in the model are examined and evaluated. The model is validated by comparing its prediction to the result of running programs on the real machines.

Chapter 4 is devoted to an analysis of the IBM 370 instruction set architecture. Some of the topics covered are: opcode distributions, instruction lengths, branch analysis, branch and execution distances, opcode pair distribution, displacement values, register use, operand lengths, memory references, and pipeline effects. The discussion includes both architectural points and results related to the specific implementations studied: the IBM 370/168-1 and the Amdahl 470 V/6.

Chapter 5 contains some similar studies of other architectures, but not to the level of detail done for the 370 in Chapter 4. The intent is to complement the 370 analysis by examining those aspects of several other computers that can contribute new information because of their difference from the 370 architecture.

Chapter 6 gives several examples of the way in which the techniques described here can be used for prediction. Examples at three different levels are presented: (1) modification to existing implementations, (2) design of new implementations, and (3) the use of this information in deriving more abstract models of computer and program behavior.

Chapter 7 contains a list of the architectural conclusions which result from this study. The emphasis is on design rules which should be followed to allow instruction sets to be implemented in very high-performance processors. Other observations about the interaction between hardware and software are included.

Chapter II
PREVIOUS WORK

There has been work in at least six areas that is related to the studies described here, or which serve to supply a perspective from which to view this work. Those areas are:

1. System-level performance evaluation
2. Benchmarking for processor evaluation
3. Modeling for processor evaluation (both analytic and simulation)
4. Measurement of language characteristics
5. Instruction timing studies
6. Measurement and evaluation of instruction types

The common thrust of these fields is an evaluation of the ultimate performance of a computer system; each approaches it from a different level or with different techniques.

2.1 SYSTEM-LEVEL PERFORMANCE EVALUATION

From the user's point of view, the performance of a computer system is a composite of the performance of the hardware, the system software, the language processors, and the application programs. The most difficult part of improving such a system is often simply identifying the bottlenecks for which modest additional work could achieve dramatic changes. A substantial literature exists which describes available techniques and examples of system performance evaluation; a recent bibliography appears in [AGA75].

Since the analysis of the total system is such a complicated affair, purchasers must rely on benchmark runs for comparative evaluation of competing systems. While this is an adequate technique for determining the overall performance of a complete computer system, it gives little useful insight about the individual components. It also encourages the attitude which blames all performance difficulties on the instruction-execution hardware because it is the obvious base. It is often much easier to improve performance by spending a few extra million dollars on a faster CPU than by considering alternative operating systems, languages, or problem-solving techniques. The work we are here most concerned with is related to the performance of the instruction-execution part of the CPU, but this emphasis should not be taken to indicate that the

other aspects of total system performance are not significant.

2.2 BENCHMARKING FOR PROCESSOR EVALUATION

Benchmarking has always been used to compare computers, but almost all such attempts result in a composite comparison of processor, I/O devices, operating system, and applications software. Some benchmark studies have been made to compare specific machines, but only when the architecture and software is the same for both is the comparison easy to interpret. Examples of this type are Amdahl 470V/6 to IBM 370/195 [SNI76] and the same Amdahl to the IBM 370/168 [EME75]. These studies, however, are concerned only with performance analysis for the user's benefit; one of the goals of this dissertation is to extend that analysis so that it is useful to the computer designer.

There has been some systematic work done to use benchmark programs for the evaluation of a variety of processors. Wichmann, for example, has used machine-independent languages to measure processing speed; in [WIC73] the time taken to execute 42 different basic statements in ALGOL 60 was measured on some 50 machines. By assuming that the time T_{ij} for a statement i on machine j can be factored as $T_{ij}=S_i*M_j$, where S_i depends only on the statement and M_j depends only on the machine, then a least-squares fitting process yields the M_j as comparative measures of machine performance.

One of the major difficulties with such an approach is that it is very sensitive to the quality of the compiler on each of the different machines. The IBM 370-series machines, for example, were at a disadvantage because the only commercially available ALGOL compiler (ALGOL-F) is known to produce rather inefficient code. Another problem is that the statements were not chosen or weighted to account for the unequal frequency of use in typical programs. In an attempt to overcome these difficulties, Curnow and Wichmann [CUR] developed a "synthetic benchmark" which has the following properties: (1) The program was made to match, as closely as possible, the same distribution of intermediate Whetstone code [RAN] as was produced by 949 programs previously collected and analyzed [WIC70]. (2) The program had to be translatable into a variety of languages (ALGOL, FORTRAN, PL/I) (3) No constructions should be used which would give a particular language or compiler an obvious advantage. For example, they attempted to "ensure that the program could not be logically optimised" so that statements would not be removed from loops by compilers clever enough to detect when it would be possible. That this had been achieved was verified by examining the object code produced.

The ranking of 10 different machines was compared using the results of the synthetic benchmark in different languages, and also compared to the ranking obtained by a

Gibson Mix (see section 2.5) and the original ALGOL statement mix. The result was that despite the care taken to avoid the influence of compiler differences, the effect of the language often made the machine differences insignificant. Two computers with one twice as fast as the other in a FORTRAN comparison could be the same when compared with the ALGOL versions of the program. The conclusion may well be that it is impossible to make language- and compiler-independent performance comparisons of machines with different architectures without simultaneously comparing the language and compilers. The study of a computer as an Algol machine, while useful for some purposes, tells little or nothing about the instruction-set processor beneath it.

2.3 PROCESSOR MODELING

Although a model of a CPU is a model of an instruction set processor, such models provide little insight into the design of the instruction set. The work in processor modeling has been mostly concerned with the internal organization of the computer, and not with representation at the level of the instruction set and the relationship to programs.

The elements of the models are the physical structures of a particular implementation -- the instruction fetch mechanism, the arithmetic units, the pipeline controls, etc.

8

The models (and the element characteristics) are sometimes simple enough to be approached analytically, but discrete simulation is more often necessary. Such studies are valuable both for understanding existing processor implementations ([NEW] is a simulation example applied to the PDP10), and for the analysis of proposed new structures ([BOW] is an example of an analytic model for multiple function-unit processors). Such modeling has not been appropriate for the analysis of instruction sets, however, since the emphasis is on the structure of the implementation and not the consequences of instruction set design.

2.4 MEASUREMENT OF LANGUAGE CHARACTERISTICS

One way to approach computer architecture is to begin with a study of the languages to be used. By determining what the common constructions are, one might then be able to design appropriate machine language primitives for efficient execution. Language studies could also be used to evaluate existing machines by examining the extent to which the frequent basic operations can be translated to efficient machine code.

Such studies have been made for several languages; among them are FORTRAN [KNU,LYO,SLA], PL/I [ELS76a,ELS76b], COBOL [SAL], ALGOL [BAT], XPL [ALE72,ALE75], and APL [SAA75]. Most of the results are concerned with the frequency of use of various source-language constructs, and

many of the results are surprising. For example arithmetic expressions, whose code optimization has received a great deal of attention, are almost always trivial; Knuth's FORTRAN study [KNU] found that 68% of all assignment statements were replacements of the form A=B, and that the average number of operands for all expressions was only 2! Such information is clearly important both for the design of languages but especially for the design of compilers.

Useful as these studies are for language research, little light is shed on the problem of matching the language to the hardware which supports it. Much of the work on the interface concentrates on compiler optimization rather than instruction set optimization. Among the few notable exceptions are the theses by Wortman [WOR72] and Hehner [HEH74]. Even there, however, the emphasis is almost entirely on the use of clever encoding techniques to reduce memory use, and the designs are therefore based on static program statistics. The result, in Hehner for example, is a 75% decrease in space compared to conventional machines, but at the cost of "an increase in hardware complexity" whose effect on performance is not considered.

2.5 MEASUREMENT AND EVALUATION OF INSTRUCTION SETS

A great deal of the literature on the evaluation of instruction sets is concerned with frequency of execution counts. The seminal work was an unpublished study in the

late 1950's by Gibson, which presented the result of measurements for the then popular IBM 7090. (The Gibson study was finally distributed a decade later as an IBM Technical Report [GIB].) In addition to instruction frequencies obtained by tracing program execution, Gibson defined 14 different basic instruction classes and their fractional use; this "Gibson Mix" has come to be used as the representation of a "standard load" for an instruction processor.

By classifying an arbitrary instruction set according to Gibson's categories, and using an average execution time for each class, the resulting instruction execution rate can be used to compare different computers. The difficulty with this approach, of course, is a consequence of its simplicity. Many instructions are hard to classify into the original categories, and variations in execution time due to address modification, operand variations, and data dependences are difficult to assess. In fact, reliable timing information for any of the high-performance computers is complex to interpret if it is even available at all. Nevertheless, some systematic uses of Gibson-like mixes have been useful in obtaining crude rankings of various dissimilar computers, especially when the mix results are presented along with results of benchmark tests, as they are in the NPL data base of computer performance information [VER]. Specialized mixes can be of use if a particular application

is involved; computer evaluation by the U.S. Army [WOL] concentrated on the solution of differential equations and developed a mix based on an analysis of the algorithms used.

A variety of data has been published for several machines since the years of the Gibson study, and many have extended the information collected to include opcode pairs, register utilization, and static vs dynamic frequency comparisons [LUN, FLY, WIN, HAN, AGA73, ANA, FOS71a,b, SAA72, ROE]. Most of these studies are unfortunately rather small in scale, due to the high cost of the data collection. In addition, little timing or performance information is provided, and no analysis of the effect of instruction-set design and use on the implementation of processors is done.

Most of the instruction-set analysis as a result of measurement has concentrated, as have the language-oriented instruction set analyis, on efficient encoding of instructions [FOS71a, HEH76, HEH77, WIL]. Although there is certainly some relationship between instruction encoding and performance, it is becoming less significant as a result of techniques such as instruction caches, pipelines, and the general increase in memory bandwidth and capacity.

A more recent systematic comparison of instruction sets has been done as part of the Computer Family Architecture study for miliary computers [FUL]. Three quantitative measures of instructions were defined: the S-measure for the

static space taken by the program, the M-measure for the amount of memory traffic generated, and the R-measure for the amount of internal processor computation required. Test programs coded for several machines were traced so that the S, M, and R measures could be determined, and a variety of programmers and programs were used so that the differences due only to the instruction set could be statistically isolated.

This approach is probably the best to date for comparing architectures independent of implementations, but it is difficult to extend its use to issues of cost/performance. It may be possible to demonstrate a correspondence between the quantitative measures and processor complexity or speed, but that work remains to be done. There are other issues with regard to the choice of test programs and the effect of higher-level languages (all test programs were in assembly language) that need to be studied further. The more important issues of what instruction types should be included at all, and what the tradeoffs are between instruction set complexity, programming ease, and processor performance have not been addressed by any of the measurement-based studies.

2.6 INSTRUCTION TIMING STUDIES

Accurate timing information for instructions and unusual conditions is required in order to be able to evaluate a processor at the instruction execution level. That information is hard to get, and the major difficulty is the lack of published data from the manufacturers, in particular for the high-performance machines. There is a distressing trend toward providing as little information as the buyer will tolerate. The excuse is often that the information is complex and difficult to present, but given the examples set by some [AMD] one cannot help but conclude that information is often withheld because it is a potential source of embarrassment. Some expurgated papers reveal glimpses of large-scale efforts with sophisticated tools [VAN, HUG, MUR], but few of the results are ever presented publicly.

This situation has forced users into the position of empirically measuring the machine characteristics; notable examples are for the CDC 7600 [LIP, MAR], and the IBM 370/168 [EME]. None of these are concerned with the relationship between the design of the instruction set and the ultimate machine performance, but they supply useful data.

The methodology involved in doing accurate timing studies for high-speed computers in real environments is complex, and some other experiences in this field were useful in developing the techniques we used [WOR76, GEN].

Chapter III

INSTRUCTION TIMING MODEL FOR MACHINE EVALUATION

One of the most important tasks for a computer designer is the evaluation of a computer architecture implementation. As a specific instance of that task, this chapter compares the implementation and performance of the IBM 370/168-1 and the AMDAHL 470 V/6, which are two high-performance machines with the same architecture [IBM70] but different implementations.

This chapter explains, in detail and by example, the methodology used for the analysis of instruction sets. It discusses the model based on instruction timing, the process of evaluating important factors, the tools needed for such a study, and the verfication procedure. Although presented in the context of the IBM 370, the approach is a general one. Results in detail for the 370 architecture and similar results for other architectures appear in subsequent chapters.

3.1 THE MODEL

The basic goal of the measurement system is to apportion the time spent by an executing program among the various system components such as the cache memory, the instruction pipeline and the individual instructions, so that resource utilization and system bottlenecks can be determined. This is achieved by using simple models of the CPU of each machine which also provide estimates of the total CPU times. The total time is important insofar as it is used to verify the accuracy of the model, since the predicted times can then be compared to the actual performance of the machines.

The decision to make implementation dependent measures of CPU performance for two members of a specific architecture family has several advantages:

1. Some of the traditionally difficult problems encountered when comparing two different architectures are not present, since many confounding factors relating to performance evaluation have the same effect on both machines.

2. The success of one of the levels of a complex system can often be measured by the characteristics of the levels below. Performance evaluation which is close to the implementation level of a computer gives valuable design information at the architecture level.

14

3. The speed of collection and the precision of the results are greatly enhanced by having tools that are tailored for a specific instruction set.

4. Practical and useful results can be obtained quickly, so that those results can be used to further refine the tools.

3.1.1 The Technique

The models of the CPUs used here are based on the instruction timing formulas available from the manufacturers' documents which describe their computers [AMD,IBM74]. These documents sometimes sacrifice details for ease of exposition (which is not to say that they are easy to read!) and represent only the best efforts of an engineer to describe the existing machine. Efforts which are described later were made to verify the accuracy of the information. In deriving the model for the Amdahl machine we were quite fortunate to get some help from the designers.

The programs to be measured were traced in user state, and all the information required to compute the instruction execution time from the formulas was collected. A record was made of counts of occurrences, values of instruction variables used in the formulas, and information about memory performance. Typical variables depend on the specific instruction but may also depend on the implementation

details. For example, the number of bytes moved is implementation independent, but measures of pipeline interlocks and timing delays are not. Some variables depend on instruction environment and therefore require information about instruction pair and triple distributions.

Two primary constraints caused us to trace only user-state instructions. (1) Tracing system software, with the attendant performance degradation of at least 50 to 1, would modify operating system behavior in timing dependent I/O sections. By tracing only in user mode, which is usually not speed dependent, we eliminate a source of error which would necessitate a complicated interpretation of the results. (2) Tracing the operating system introduces a large number of problems involving the recording of the trace data. One standard solution is the use of samples rather than complete traces, but then the verification of the predicted CPU time is not possible.

Since the timing formulas do not include the effects of cache memory misses, the cache memory is simulated for each machine. The cache miss penalty is added to the instruction execution time to obtain the expected program execution time. To verify the model the expected time is compared to the operating system accounting time corrected to compensate for the differences between the measurement methods.

The effects of instruction interaction, which can generally be attributed to pipeline resource interlocks, are rather explicitly accounted for in the Amdahl formulas. For IBM, however, the pipeline effects have been averaged into the formulas in a way which was not clearly indicated. This was a potential source of difficulty, but the effort required to obtain this information from the logic diagrams and microcode listings was prohibitive, and unjustified when an error of a few percent is acceptable.

The techniques used here are much more complex than benchmarking, but not as costly as total hardware simulation. The tools are general enough so they can be -- and have been -- used for other studies with different objectives. The importance, however, lies in the ability to change the model variables to reflect proposed changes to the existing hardware and to accurately predict the performance effects of those changes.

3.1.2 Evaluation of Important Factors

The development of the CPU model has been greatly influenced by the idea of an evolving system of tools -- development by successive refinement. A crude model and simple tools were first assembled and by successive iteration new tools, new measurements, and a more refined model were designed. This approach reduces the number of false starts and the elapsed time of the whole study by

quickly allowing the effort to concentrate on the most important factors.

The CPU model used is an intermediate one between full simulation at the hardware register level and a machine-independent representation of performance. The decision to include some factors and exclude others was based on an estimate, often supported by experimentation, of the effect of those factors on the final results. Some of the justification for the decisions are presented below.

The accuracy of the model is supported by the match between the program execution time as predicted by the model and the same time measured by the operating system during actual runs. Performance evaluation by benchmarking is repeatable only within 2-3% because of the large number of uncontrollable variables, and this therefore puts an upper bound on the precision of the validation.

An examination of previously published instruction frequencies might suggest that the more frequent instructions are those whose duration is constant and therefore do not heavily depend on execution variables like the length of operands. If this were true, then those variables could be set to program-independent values without introducing a significant error in the result. To test this hypothesis, the program which computes execution times was given three sets of execution variables with which to predict program

running time. One was a programmer's best guess of the true values, and the other two were the smallest and largest extremes which could realistically be expected. The results showed that an instruction could jump from 4% to 50% of the total time depending on the value of its variables with all others remaining the same. This is an unacceptable error, especially since errors in the variables for many instructions could combine to form large systematic errors. Most of the variables which affect execution time were therefore measured exactly or estimated from related measurements.

The predicted execution time is composed of the aggregate instruction timing results and a penalty for cache memory misses. The aggregate instruction timing results have already taken into account the instruction counts and basic execution speed, as well as the pipeline interlocks. The cache miss penalty depends on the reference pattern of the program, the cache organization, and the data flow pattern within the machine. The two machines differ rather markedly in those respects: the 370/168 uses aligned doubleword (8-byte) accesses and an associative set size of 8, while the 470 accesses unaligned fullwords (4-bytes), uses a set size of 2, but has the same total amount of data (16K bytes). Both caches use an LRU algorithm to determine which 32-byte line within a set is replaced when new data is to be fetched[1]. There are also significant differences in

the amount and type of instruction lookahead performed. To accurately measure the cache penalty, the trace analysis program has a detailed simulation of the cache and instruction fetch mechanism of both machines.

Although cache memory miss ratios are known to be low [MER], it is easily shown that the contribution of the time penalty for the misses is too large to be neglected. If the miss ratio is 5%, with a 480 nsec penalty for a miss, 1.6 memory requests per instruction, and an average instruction execution time of 300 nsec (reasonable values for the 370/168) then the cache misses represent a 13% increase in the execution time.

Two other cache organization features must be considered in the cache penalty correction. For IBM, stores always access main memory ("store-through") which may cause extra delays. For Amdahl, there is an extra penalty when a 4-byte access crosses a cache line boundary. These and the other cache corrections are not attributed to the instructions which caused them, but rather accumulated separately.

----

[1]Although the 168 set size is 8, the LRU algorithm treats each pair of two lines as a group and the least recently used group is replaced. This is "worse" than a true 8-way LRU algorithm since very old data may be paired with new data in a group and never be replaced, but the cache was originally designed for 8K and was expanded with minimal hardware additions.

The measurement of user-state instructions in a production operating system environment caused some difficulty in determining the true CPU time. The execution time reported by the operating system includes all user-state and some supervisor-state instructions [BEN], whereas the trace program measures only user-state instructions. The time attributed to these supervisor-state instructions executed in the processing of user-initiated supervisor calls (SVCs) must therefore be subtracted from the reported CPU time. Measurements were made of the charged time for all the relevant SVCs as the programs were traced. The correction is very significant for almost all programs, since both the number and cost of the SVCs are high. For the 168, for example, the time charged varies from 107 usec for an I/O operation to 26 msec for opening a file.

Although the SVC time correction could have been measured for the original benchmark programs, they were somewhat modified in view of the substantial correction required (as much as 20%). Wherever possible, the number of I/O operations was reduced by increasing the file blocking factors, but the operation of the programs was not otherwise altered. Despite this effort, the SVC time correction remained the factor which introduced the largest error in the measurements. We also traced a FORTRAN numerical analysis program from which the I/O parts were excised, so that few supervisor services were requested.

Since supervisor-state and user-state instructions share the same cache, there will be some displacement of the user's "working set" from the cache in response to an SVC, which will manifest itself as a lower than normal hit ratio when the user's program is resumed. An unpublished note by Rossman suggested that this would have a large effect [ROS]. To verify this the cache activity for one job with a large number of SVCs was simulated -- first assuming a 100% cache flush for each SVC, and then again with no flush. The number of cache misses changed by a factor of 10. Measurements showed that the actual fraction of the cache displaced by the various SVCs varies from 0.16 to 1.0, and that almost all non-trivial requests completely replace the cache. The cache simulator therefore took this into account by flushing the appropriate fraction of least-recently-used cache entries when an SVC was executed.

Interrupts which occur during the execution of the program do not account for a significant increase in accounted time (since the user-state CPU timer is disabled during interrupt processing) but there could be an effect due to cache displacement caused by the interrupt routine. On a heavily loaded machine interrupt rates as high as 4000 per minute are common, representing at worst 16.4 ms of extra time (1.7% for IBM) to completely refill the cache for each second of CPU time. Since most of those interrupts are due to other jobs, this effect was reduced to a negligible

level by running the job on on otherwise idle system, so that only the few interrupts caused by the benchmark job itself could cause interference. This is unlike the SVC correction, for which no change in the number of cache flushes is possible simply by controlling the environment of the benchmark run. Similar calculations for the effect of channel I/O transfers to memory show that they have even less effect on CPU performance. This is true both for IBM, where the channels transfer directly to main memory and invalidate corresponding cache entries, and for Amdahl, where the channels transfer into the cache.

3.1.3 Instruction Timing Formulas

An instruction may have several timing formulas associated with it, corresponding to different modes of execution. Each individual timing formula may depend linearly on the variables (the most common case) or have a more complicated dependence. In general, three types of linear formulas are encountered.

Some timing formulas reduce to a constant, and often only one formula is associated with an instruction. Examples of this case are most register-to-register arithmetic or logical instructions.

ADD REGISTER (AR)	IBM	.080 usec
	Amdahl	.065 usec

Many formulas have a simple linear dependency on execution variables. An example is a Load Multiple (LM) instruction which can be expressed as

Load Multiple (LM)	IBM	.520+.080*R usec
	Amdahl	.065+.065*R usec

where R is the number of registers loaded.

Some formulas may involve variables which are concerned with the general environment of the instruction. These are often measures of the effect of pipeline interference which causes a delay in the execution of an instruction. Examples are the Amdahl variables S1 and DWD. S1 accounts for some cases of pipeline interlocks, and ranges from 0 to .065 usec depending on the "number of execution cycles attributable to the three words of the instruction stream following the instruction of interest" [AMD]. DWD, which is either 0 or .0325 usec, compensates for the occurrence of a doubleword result instruction before the subject instruction, because the machine is fundamentally single word oriented.

Store (ST)	Amdahl	.065+S1+DWD

When several formulas are associated with one instruction, each formula applies only to a specific case of its execution. For example, the Move Character instruction execution formulas depend in important ways on the degree of overlap of the two operands. The different cases involve not only different coefficients, but often different variables.

19

```
Move         IBM    .760+.040*B usec       (no overlap)
Character           .640+.240*B usec       (any overlap)
(MVC)

         Amdahl .195+S1+.130*WB+MV usec

          where  MV = .130*W     (no overlap, or
                                  overlap>32 bytes)
                 MV = .1625*W    (3<overlap<=32 bytes)
                 MV = .130*B     (1<overlap<=3  bytes)
                 MV = .195*B     (overlap=1 byte)

       and where  B = number of bytes moved
                  W = number of words moved
                 WB = number of bytes which must be
                      moved to have the destination
                      field on a word boundary when B>63.
```

For all the individual linear formulas, only the counts and average variable values for each of the timing formula cases need to be accumulated.

Unfortunately, some formulas are not linear in their variables. Typical examples are the decimal arithmetic instructions, where the duration depends on the product of the lengths or the average value of the digits used. For these the appropriate products of variables are accumulated at the time the program is analyzed, and these values are averaged for use by the other programs in an equivalent linear form. These cases of non-linear formulas are sufficiently infrequent to justify this special treatment, but the effect on timing values is too important to ignore them. A simpler approach would assume that the product of the averages is a sufficient estimate of the average product, but the potential error is great.


```
Divide       IBM    2.42+.60*(N1-N2)*N2
Decimal
(DP)        Amdahl  20+(5+4*Q)*(N1-N2)+2*RW  if N2<=3
            Amdahl  27+(9+6*Q)*(N1-N2)+2*RW  if N2>d3

          where  N1 = length of first operand
                 N2 = length of second operand
                  Q = 1 + average value of the quotient
                      digits
                 RW = number of fullwords in the result
```

The formulas are encoded as a string of records, each corresponding to the coefficient of a term in a subcase of a timing formula for a particular instruction; there are a total of 3200 variable names and coefficient values. A numbering and naming scheme was devised that allows variables which are common to many formulas to be propagated to all appropriate places, as well as giving individual identities to variables which are used more restrictively.

3.1.4 Measurement Approximations

Some of the variables required for the computation of the instruction times are difficult to measure; this is especially true for those which involve interactions of sequences of intructions. An approximation technique was used for the computation whenever it could be shown that it would not significantly degrade the accuracy of the result.

As an example of this approximation technique, consider the interlock variables called S1, S2, and S3 for the 470, which represent time penalties added to an instruction

20

because the instructions which follow do not have enough execution cycles to "hide" the completion of the subject instruction. The time penalty caused by these pipeline "breaks" depends on a particular sequence of instruction types, but can be more simply described as an addition to the execution time of particular instructions whenever they are in certain environments. For the Sn interlocks, the definitions are:

```
S1 = 0  if  M = 8
     1  if  M = 7
     2  if  M = 6
```

where M is the number of execution cycles attributable to the three words of the instruction stream following the subject instruction, and

```
S2 = 0  if  P = 8          S3 = 0  if  P = 6
   = 1  if  P = 7             = 1  if  P = 5
   = 2  if  P = 6             = 2  if  P = 4
   = 3  if  P = 5
   = 4  if  P = 4
```

where P is the number of execution cycles attributable to the two words of the instruction stream following the instruction of interest.

In order to avoid computing the frequency of occurence of all 7-tuples of instructions (at most 6 instructions can follow the subject instruction in the three words which follow), a set of instruction classes was defined where each class contains instructions of the same size and approximately the same execution time. It is then computationally

feasible to accumulate the occurence of relatively long sequences of classes and use these to estimate the true value of the variables. For each instruction which incurs any of the Sn penalties, all possible sequences of subsequent classes are examined. For each sequence, classes are examined until the sum of the instruction lengths exceeds three words (for S1) or two words (for S2 and S3). The sum of the average execution time of the instructions of each class in the sequence is then used as the value of M (for S1) or P (for S2 and S3). Each possible sequence is weighted by its probability, estimated by the frequency of occurence.

The characteristics of the 10 classes were chosen to minimize the error in the computation. All instructions with the same length which have execution times greater than 6 cycles are put in the same class, since the occurence of any instruction of this type in the sequence will automatically force the interlock parameter to be zero. The other classes make much finer distinctions between instructions with small execution time, since they have the potential of making a whole cycle difference in the final value.

Note that the class assignment depends on the estimated execution time, which in turn depends (because of the values of the interlock variables) on the class assignments. Initial class assignments were based on rough estimates of

execution time using what seemed to be reasonable guesses for interlock variables. The class assignments were refined as a result of getting more accurate execution times but the differences were negligible because the averaged value of the interlock variables is so small that it has a very weak effect on execution times and hence made almost no difference in class assignments.

3.1.5 Description of the Measurement System

The heart of the measurement system is an interpretive trace program (TRACE) which executes arbitrary load modules (relocatable program files) containing user-mode instructions. Programs are interpreted with precisely the same environment that would be experienced without the tracer; the real data files are used, the same supervisor services are requested, and the same results are produced. As a side effect, however, instruction trace records are produced which can be used to form a detailed characterization of the program. Other trace records are produced which contain information relevant to the tracing process, or to program activities such as exception interrupts.

The record produced by the interpreter for each instruction contains the instruction type, memory addresses referenced, and the other required information. These records are processed by a trace analysis program (ANALYSIS) which generates instruction counts, variable values, and

memory access statistics such as cache memory miss counts, which are stored in a summary file. In order to avoid saving massive amounts of intermediate trace information (25 megabytes per traced second), the TRACE and ANALYSIS programs may execute as coroutines. The combined overhead of the trace and trace analysis programs amounts to 300 seconds per second of real time. This compares favorably to other more detailed hardware simulations, where the overhead has been as high as 6000 seconds per second of real time [VAN].

The summary file is converted into a count file by an intermediate program (CONVERT). The count file contains all the information required to compute the timing formulas for both machines (the 168 and the 470) condensed into about 600 numbers. An instruction statistics program (INSTAT) uses the count file and files of encoded instruction timing formulas to produce the timing and performance information.

Several test programs were devised for verifying the formulas and understanding the measurement factors. A general instruction timing program (LTIMER) was designed for precise measurements of instruction times, cache memory miss penalties, SVC times, and the effects of SVCs on cache memory contents. Each instruction, instruction sequence, or memory access pattern can be executed in a variety of environments so that a systematic study of the variations in execution time can be made.

22

The measurement system therefore consists of a number of independent programs which communicate through well-defined interfaces, as shown in Figure 1. Each of the interfaces is general enough so that other programs could have been used -- and in fact at various times were used -- for other types of processing. A number of address traces for other cache and virtual memory simulations have been produced [RAF,SMI,YU]. We could have merged some of the processing programs, but in doing so would have lost some generality. By requiring that generality we have guaranteed, for example, that neither the trace program nor the trace analysis program knows anything about the details of the timing formulas. In fact, only CONVERT has specific knowledge of them; INSTAT knows only about the timing formula description convention, and in that sense is independent even of the general architecture of the machine being studied.

Writing an interpretive trace program is basically a straightforward exercise, but there are two classes of problems. The first are a consequence of the hardware architecture, and may require some awkward but tractable solutions. The requirement for having a base register for the trace program, for example, is often made difficult by instructions which can potentially modify all registers, such as Load Multiple. The second and more serious class of problems are a consequence of the software architecture.

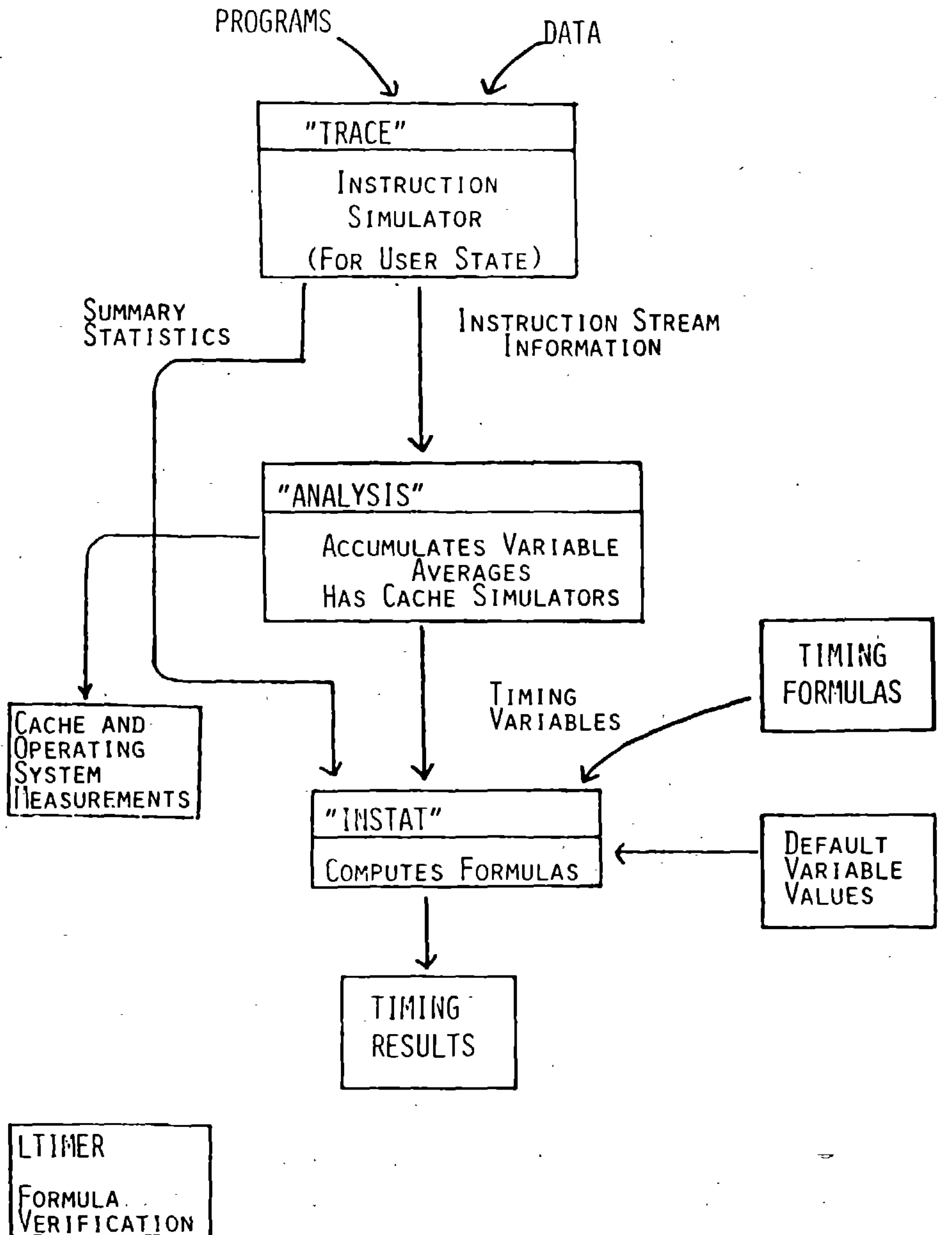


Figure 1: Measurement System

23

The manufacturer's hardware manual is a complete and accurate description of all instructions except one: Supervisor Call (SVC). The operating system makes SVC a difficult and unpredictable instruction to trace, primarily because of the variety of cases in which control is not transferred to the instruction following the SVC. Fully half of the trace interpreter is devoted to problem SVC's, such as LINK (dynamic transfer to a relocatable program file), SPIE (create a program-interrupt environment), EOV (test for unusual file conditions), and many others. It would be to the great benefit of many computer users if the designers of software were required to be as accurate and complete in the description of their systems as the designers of hardware.

3.2 VERIFICATION

This section explains the procedure for verifying cache miss and SVC times, describes the benchmark jobs used, and shows the computation of predicted times and comparison with benchmark runs.

3.2.1 Cache Miss Penalty

Although cache miss penalty information is available from the manufacturers, it was difficult to interpret precisely what the effect on instruction time is. Since measurements are not difficult and the correction could be

significant, the values were verified experimentally. To determine the cost of a cache miss, a test program simply fills the cache with known data. A second loop is then timed, in which either the same data is reloaded, or new data displaces the old. The difference in time between the two versions of the second loop, divided by the number of cache misses caused by the loop which displaces the data, provides the cache miss time. The value found for IBM is 480 nsec, which is not inconsistent with the (rather confusing) information from the hardware manuals. For Amdahl, cache misses are found to cost 650 nsec, which also agrees with information from the designers.

Once the cache miss penalty is established, the effect of a supervisor request on the user data in the cache can be measured easily. In a similar fashion the cache is filled with known data, the SVC is issued, and the cache is refilled with the same data. The second loop is timed, and compared to the identical loop when the SVC is not present. The time difference divided by the cache miss penalty gives the number of cache lines that were displaced by the SVC. Note that the second loop must fill the cache in the opposite order from the first loop, otherwise the LRU replacement algorithm would cause the original data to be removed instead of the data added by the SVC. Table 1 shows the fraction of cache displacement for some of the more common supervisor requests.

Table 1.

SVC Times and Cache Requests

(Averaged for all programs)

Name	IBM CPU time usec.	IBM % cache displaced	Amdahl CPU time usec.	Amdahl %cache displaced
OPEN	26658	100%	17605	100%
CLOSE	16929	100%	13488	100%
EXCP I/O	107	58%	101	24%
WAIT	234	16%	139	7%
REGMAIN	394	30%	219	17%
LINK	3629	100%	1613	41%
OVERLAY	5214	100%	N/A	N/A

One of the interesting differences of implementation between the two machines is the effect of data stores on the cache. The IBM approach is to always store data directly into main memory, and to update the cache only if the line already exists. The Amdahl machine updates the cache line if the data is present without storing into main memory. If the data is not in the cache, the line will be read from memory. If the replacement algorithm must remove a line which was modified in the cache, the memory is updated at the time the line is replaced. The IBM method, called "store-through", has often been criticized because it requires a main memory access for all stores [KAP]. Although the store can proceed in parallel with subsequent instructions, any subsequent main memory accesses must be suspended until the memory becomes available. Since the timing formulas do not explicitly account for this effect, it is important to determine its magnitude.

There are three factors which combine to minimize the possible deliterious effects of the store-through policy used by IBM. The first is that the memory is organized with four-way interleaving of adjacent doublewords, so that consecutive stores may well reference separate memory banks. The second is simply that based on the opcode pair distribution we have accumulated, consecutive instructions which store data into memory are relatively infrequent. The third is that even for pairs of such instructions, there is a level of buffering for data that is to be written to main memory, at least for the case when that data is also in the cache. A penalty appears only for the third consecutive store, and then is 360 nsec. The full write cycle time penalty of 640 nsec occurs only for the fourth and subsequent store. These factors are sufficient to justify not including a correction for store-through writes.

3.2.2 SVC Times

As previously discussed, the CPU time charged for SVCs was measured in order to be able to correct the time given by the operating system. The time charged for each SVC is often large and varies from program to program even for the same SVC type. To account for these variations the time charged to the user for each SVC was measured as the benchmark programs were being traced. The SVC correction computed by summing the measured SVC times is therefore

quite accurate for the 168 because it was the machine used for the tracings. For the 470, the timing program LTIMER was used to give estimates of the average SVC costs. This latter method does not take into account the variation from program to program and the SVC corrections are much less accurate than for the 168. Table 1 shows the time charged for some important SVCs averaged over all programs.

It is interesting that the time charged for supervisor services is often comparable to what would be required if there were no operating system. For I/O operations, previous measurements have shown that the hardware I/O instructions (SIO, TIO, etc.) are incredibly expensive; 100 usec is not unusual [JAY]. This is to be compared with, for instance, the measured charge of 107 usec for the request to the operating system for an I/O operation. Note that both of these are more than two orders of magnitude larger than, for example, the 0.61 usec needed for a double precision floating point multiplication. It would seem that improvements in the arithmetic units of computers have not been accompanied by similar improvements in the I/O interface despite the existence of I/O channels.

3.2.3 Benchmark Jobs

The results presented here are derived from the complete analysis of seven benchmark jobs written at the Stanford Linear Accelerator Center[2]. Except for one

(LINSY2) they were all production jobs written for purposes other than performance evaluation. To avoid biasing the results with artifacts from specific languages or programs, we purposely chose the three most used language compilers and programs compiled by them.

1. FORTC is a compilation by the IBM Fortran-H optimizing compiler.
2. FORTGO is the execution of the FORTRAN program compiled by FORTC. It is a numerical analysis program which solves partial differential equations.
3. PL1C is a compilation by the IBM PL/I-F compiler.
4. PL1GO is the execution of a PL/I program which accumulates and prints accounting summaries from computer use information.
5. COBOLC is a compilation by the IBM ANSI Standard COBOL compiler.
6. COBOLGO is the execution of a COBOL program which reformats and prints computer use accounting information.

---

[2]Dozens of other jobs were traced to provide other data presented elsewhere, but only these seven were fully analyzed and run in carefully controlled benchmark conditions on both machines. Several hundred million instructions were traced and analyzed in the course of this study.

7. LINSY2 is the execution of a FORTRAN subroutine which solves large-order simultaneous equations. No I/O is done.

Table 2 summarizes some characteristics of the benchmark jobs.

Table 2.

Program Characteristics

Program	# Instr.	Data reads per inst	Data writes per inst	Inst/Cache Miss IBM	Inst/Cache Miss Amdahl
COBOLC	6,048,476	0.431	0.130	82.57	36.95
FORTGO	23,865,168	0.352	0.204	104.06	28.07
PL1GO	23,863,497	0.473	0.261	73.28	61.16
LINSY2	11,719,853	0.195	0.067	20597	19598
COBOLGO	3,559,533	0.738	0.453	13.42	30.93
FORTC	17,132,697	0.433	0.146	39.86	24.47
PL1C	24,338,101	0.379	0.137	145.33	63.48

3.2.4 Validation

Verification basically consists of comparing the time predicted by our model for each benchmark job with the corrected real execution time. The time predicted for each benchmark, Tpred, consists of the following terms:

Tins, the total time predicted from the timing formulas, which does not include the cache miss penalty.

M * Tmiss, where M is the number of cache misses as reported by the cache simulator, and Tmiss is the cache miss

penalty. The number of cache misses includes the effect of SVC execution on the cache contents.

Tcross, the time penalty, for Amdahl only, paid when references to the cache cross a line boundary. The penalty is two cycles (.065 usec) for reads and three cycles (.0975 usec) for writes, and is computed using numbers provided by the cache simulator. Virtually all the penalty arises from instruction fetch, since none of the programs access unaligned data. There is no equivalent penalty for IBM because its larger instruction buffer prefetches enough so that two successive doublewords can be accessed without introducing an additional delay.

The corrected time for the actual execution, Trun, consists of the following terms:

Tacc, the time as given by the standard IBM accounting routines.

-Tsvc, the time attributed to the user for the execution of all the supervisor calls, which must be subtracted from Tacc.

Table 3 provides the values for each of these times for each of the benchmarks. For Tpred and Trun, the relative percentage of each of their components is given. The absolute error, Trun-Tpred, and the percent error, (Trun-Tpred)/Trun, appears on the last lines. The verification process points to large discrepancies between the basic execution time of instructions (Tins) and the speed as perceived by the user (Tacc).

There are a variety of methodological problems involved in accurate benchmarking. Because these were real programs doing I/O to real devices and interacting in complex ways with the operating system, it was very difficult to get accurate and repeatable measurements. In order to achieve accuracy to within about 1%, it was necessary to carefully control running conditions: using otherwise idle time at night (the computers are normally multiprogrammed), removing operating system variations by specific device allocations, memory control to isolate paging variations, etc. Each program was run many times, and the resulting averages were used only if the deviation with particular set of conditions was small; these same conditions were then used during the tracing phase. The goal of repeatable measurements was achieved, but the difficulty of doing so was rather more than expected.

Table 3.

Predicted and Benchmark Times

COBOLC	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	2.213	98.44	1.179	88.45	1.878
M*Tmiss	.035	1.56	.106	7.95	.330
Tcross			.048	3.60	
Tpred	2.248	100.00	1.333	100.00	1.686
Tacc	2.57	100.00	1.71	100.00	1.503
-Tsvc	.348	13.54	.320	18.71	1.088
Trun	2.222	86.46	1.390	81.29	1.599
Trun-Tpred	-.026		-.057		
% error	-1.170		-4.101		

FORTGO	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	6.176	98.25	3.286	83.81	1.879
M*Tmiss	.110	1.75	.553	14.10	.199
Tcross			.082	2.09	
Tpred	6.286	100.00	3.921	100.00	1.60
Tacc	6.42	100.00	N/A		
-Tsvc	.082	1.28			
Trun	6.338	98.72			
Trun-Tpred	.052				
% error	0.82				

PL1GO	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	4.561	96.69	2.233	85.88	2.042
M*Tmiss	.156	3.31	.254	9.77	.614
Tcross			.113	4.35	
Tpred	4.717	100.00	2.600	100.00	1.814
Tacc	5.45	100.00	3.42	100.00	1.594
-Tsvc	.293	5.38	.206	6.02	1.422
Trun	5.157	94.62	3.214	93.98	1.604
Trun-Tpred	.440		.614		
% error	8.53		19.10		

LINSY2	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	1.970	100.00	1.561	96.48	1.262
M*Tmiss	.000	0.00	.000	0.00	1.000
Tcross			.057	3.52	
Tpred	1.970	100.00	1.618	100.00	1.218
Tacc	1.98	100.00	1.69	100.00	1.172
-Tsvc	.040	2.02	.031	1.83	1.290
Trun	1.940	97.98	1.659	98.17	1.169
Trun-Tpred	-.030		.041		
% error	-1.55		2.47		

COBOLGO	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	4.291	97.13	2.451	95.67	1.751
M*Tmiss	.127	2.87	.075	2.93	1.693
Tcross			.036	1.40	
Tpred	4.418	100.00	2.562	100.00	1.724
Tacc	4.82	100.00	2.92	100.00	1.651
-Tsvc	.428	8.88	.289	9.90	1.481
Trun	4.392	91.12	2.631	90.10	1.669
Trun-Tpred	-.026		-.069		
% error	-0.59		2.62		

FORTC	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	3.711	94.74	1.886	77.62	1.968
M*Tmiss	.206	5.26	.455	18.72	.452
Tcross			.089	3.66	
Tpred	3.917	100.00	2.430	100.00	1.612
Tacc	4.64	100.00	3.10	100.00	1.497
-Tsvc	.652	14.05	.430	13.87	1.62
Trun	3.988	85.95	2.670	86.13	1.494
Trun-Tpred	.071		.239		
% error	1.78		8.95		

PL1C	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	7.372	98.93	3.846	88.94	1.917
M*Tmiss	.080	1.07	.250	5.78	.320
Tcross			.228	5.27	
Tpred	7.452	100.00	4.324	100.00	1.723
Tacc	8.16	100.00	4.93	100.00	1.655
-Tsvc	.794	9.73	.388	7.87	2.046
Trun	7.366	90.27	4.542	92.13	1.622
Trun-Tpred	-.086		.218		
% error	-1.17		4.80		

The results for IBM are generally extremely good; for all except one program the differences between the predicted and actual running time are less than 2%. The agreement for Amdahl is not as good, but most of the error can be traced to the crude method for measuring the SVC time correction. A factor of two in the the SVC correction, which is certainly conceivable when an OPEN as measured on the 168 can vary from 6 to 33 msec, can easily account for all the error. For both machines, the programs for which the match was the poorest (such as PL1GO) were those that had high I/O activity and therefore required the largest SVC correction.

Chapter IV

CHARACTERISTICS OF THE IBM 370 ARCHITECTURE

This chapter contains the results of measurements for the 370 which were produced by the procedures described in Chapter III. The results are often presented in a largely descriptive fashion; more information by way of conclusions and summary appear in Chapter 7. Similar results for other architectures appear in Chapter 5.

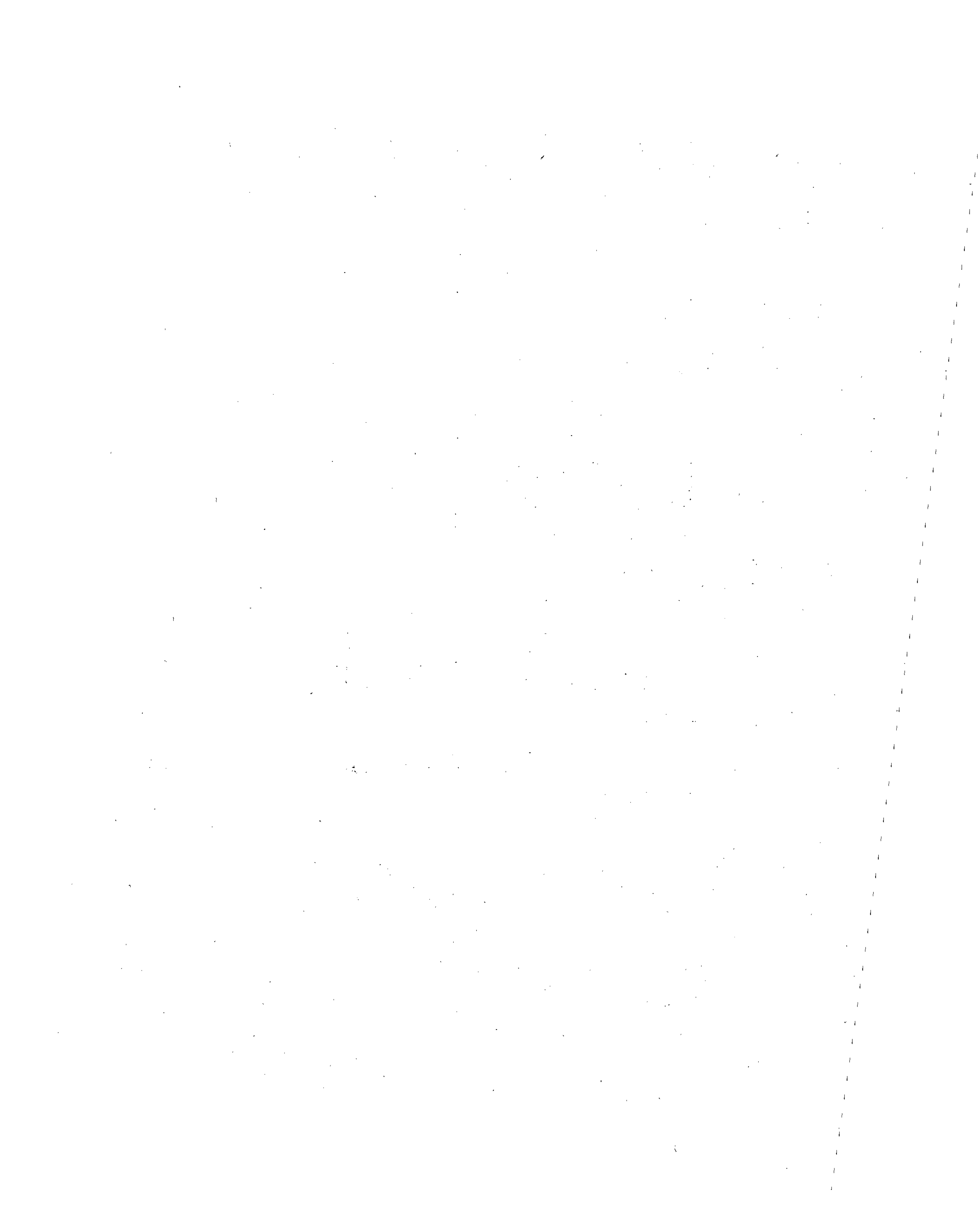
4.1 OPCODE DISTRIBUTIONS

Figures 2 through 11 show opcode distributions for a variety of different programs, including all the benchmark programs discussed in Chapter 3[1]. The "Static Count" column shows the frequency of occurence of opcodes as they occur in the program text. The "Dynamic Count" column represents counts of instructions executed when the program was run. The two "Time" columns show how the execution time was distributed among the various instructions for the 168 and the 470 computers.

---

[1] Appendix A contains a brief explanation of the instruction mnemonics for the 370 and the other machines examined.

As has been observed many times, very few opcodes account for most of a program's execution. The COBOLC program, for example, executes 84 of the available 183 instructions, but 48 represent 99.08% of all instructions executed, and 26 represent 90.28%. Note that it is common for an instruction to have a ratio of 1 to 5 in execution time percentage versus execution frequency. For example, the "Move Chararacter" (MVC) instruction in the COBOLC job represents 3.92% of all instructions executed, but accounts for 14.97% of IBM execution time, and 16.47% of the Amdahl execution time. In contrast, the "load" (L) instruction in the COBOLGO job represents 16.58% of all instructions executed, but accounts only for 1.65% of IBM execution time, and 1.57% of Amdahl execution time.



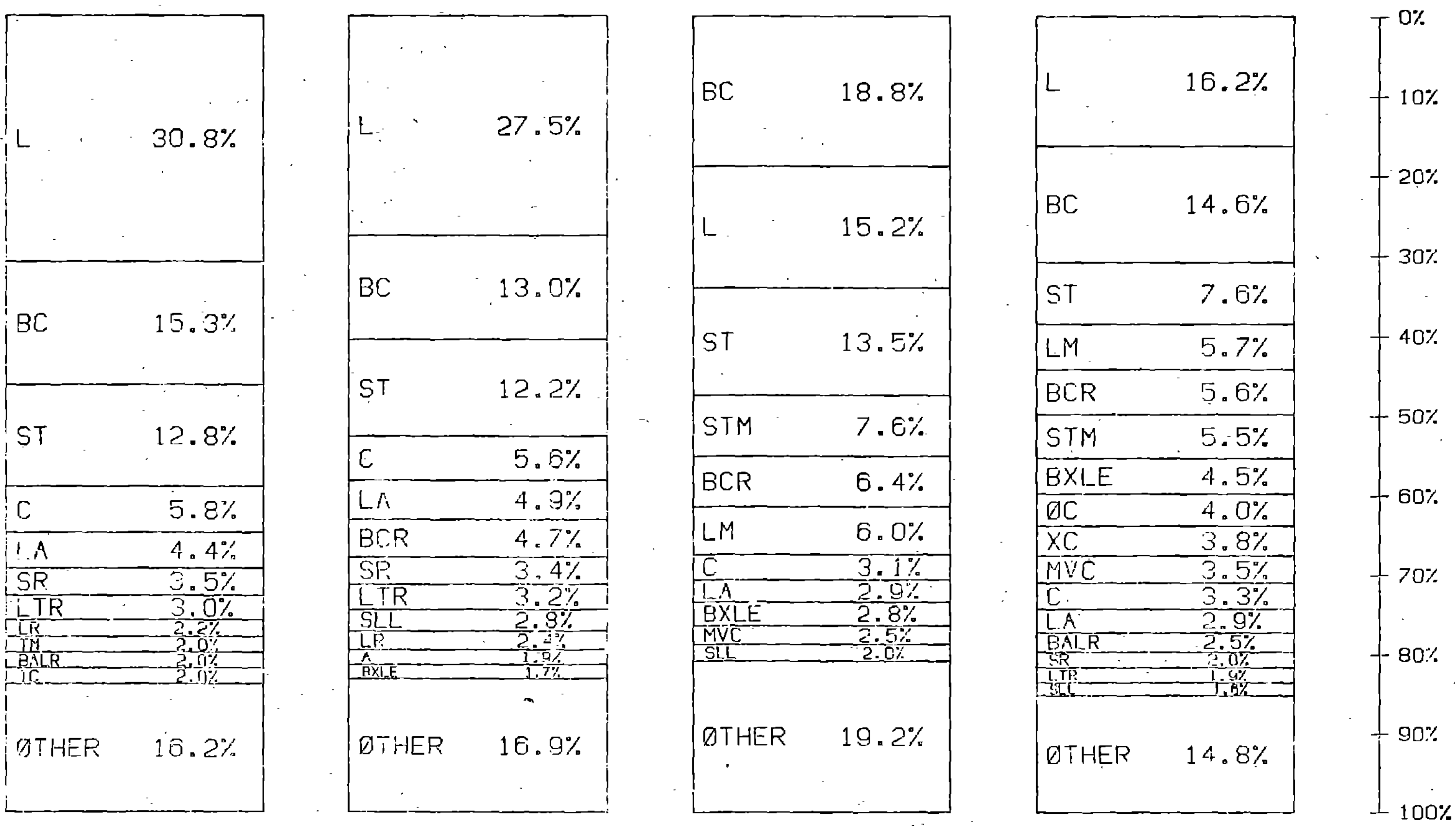
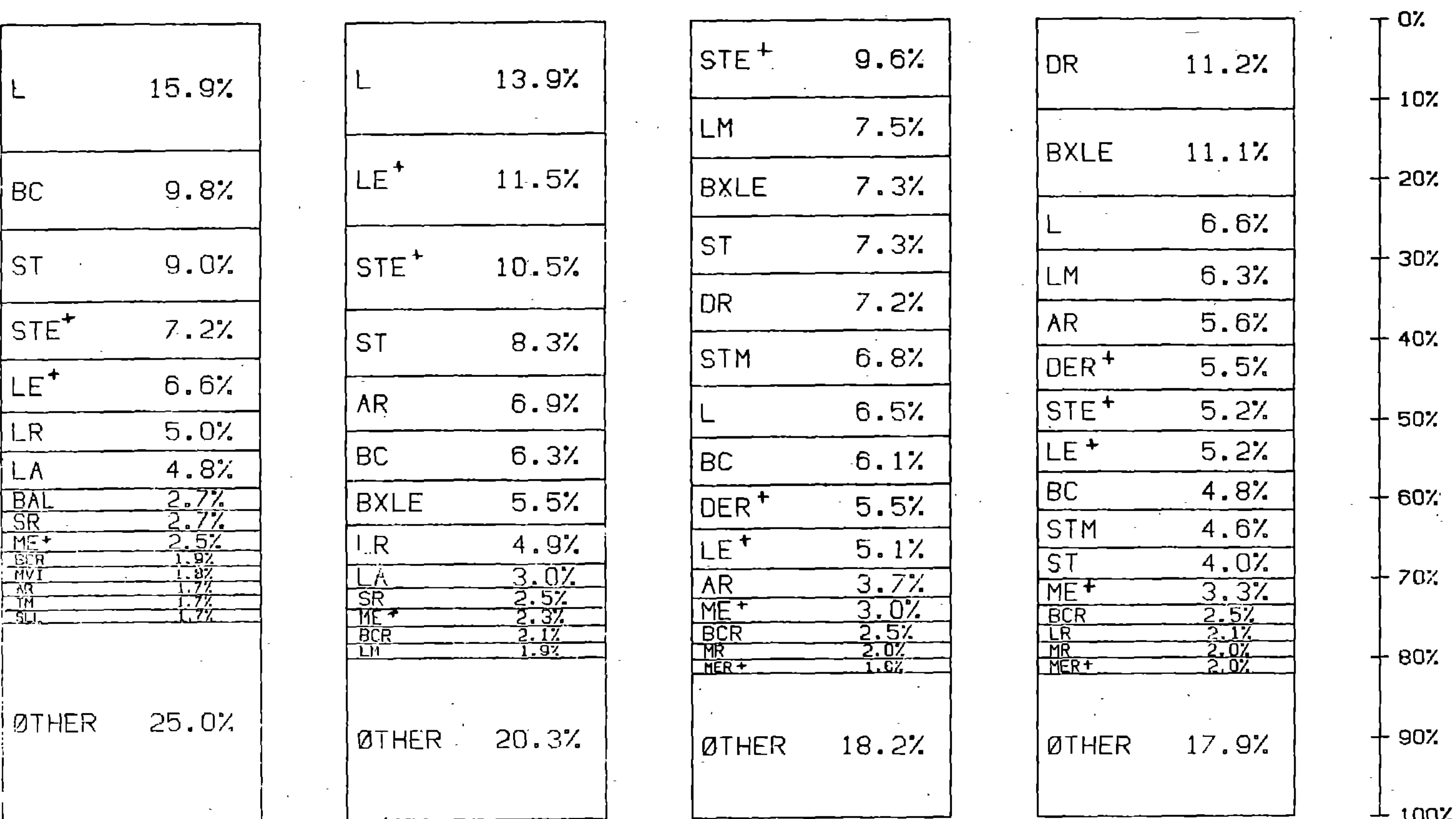


Figure 2: Static/Dynamic Count, and Time by Opcode - FORTC



+Floating Point Instruction

Figure 3: Static/Dynamic Count, and Time by Opcode - FORTGO

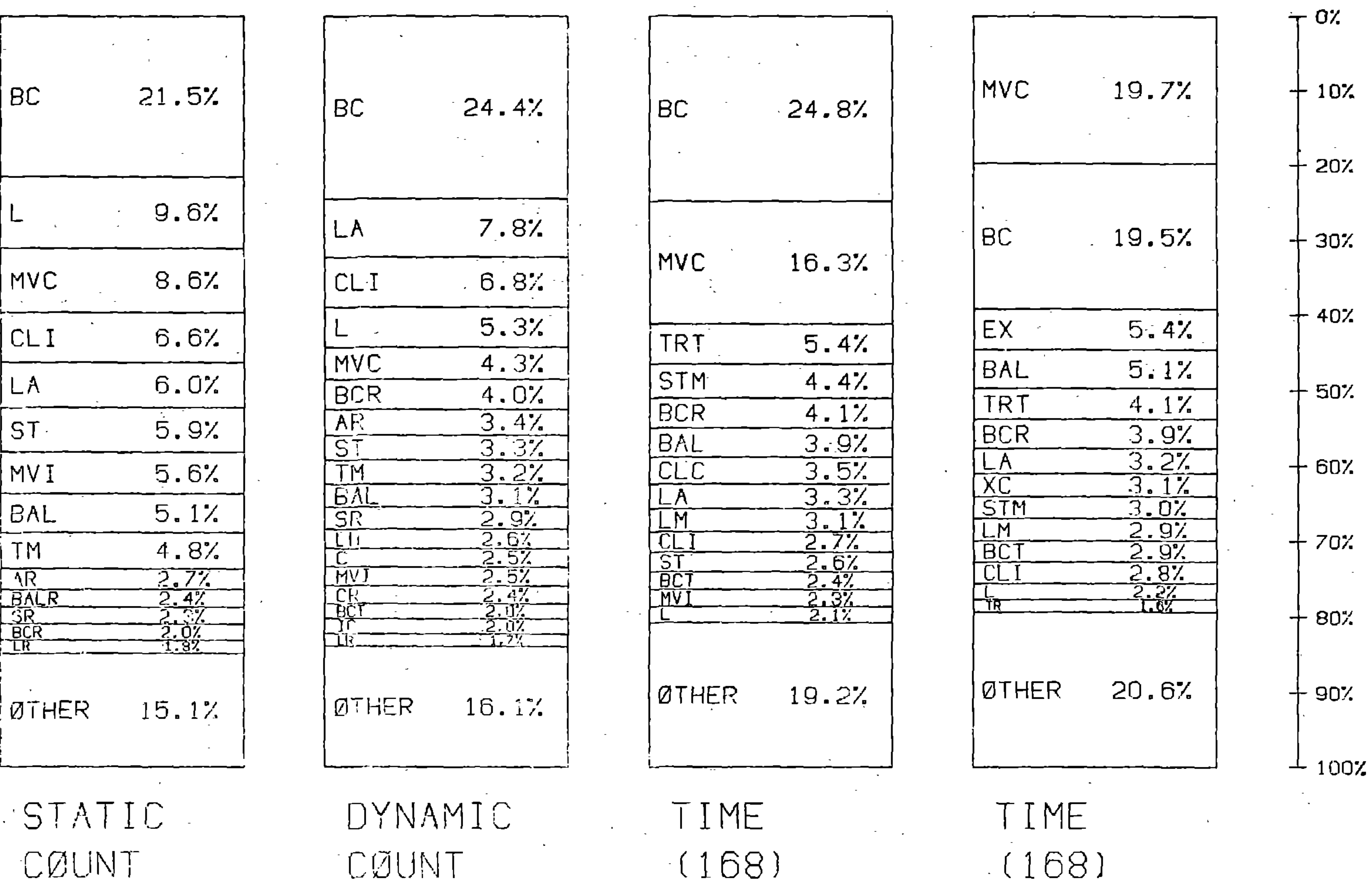


Figure 4: Static/Dynamic Count, and Time by Opcode - PLIC

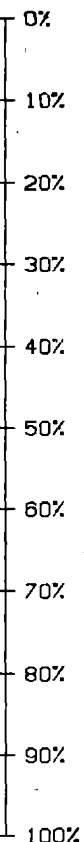


Figure 5: Static/Dynamic Count, and Time by Opcide - PLIGO

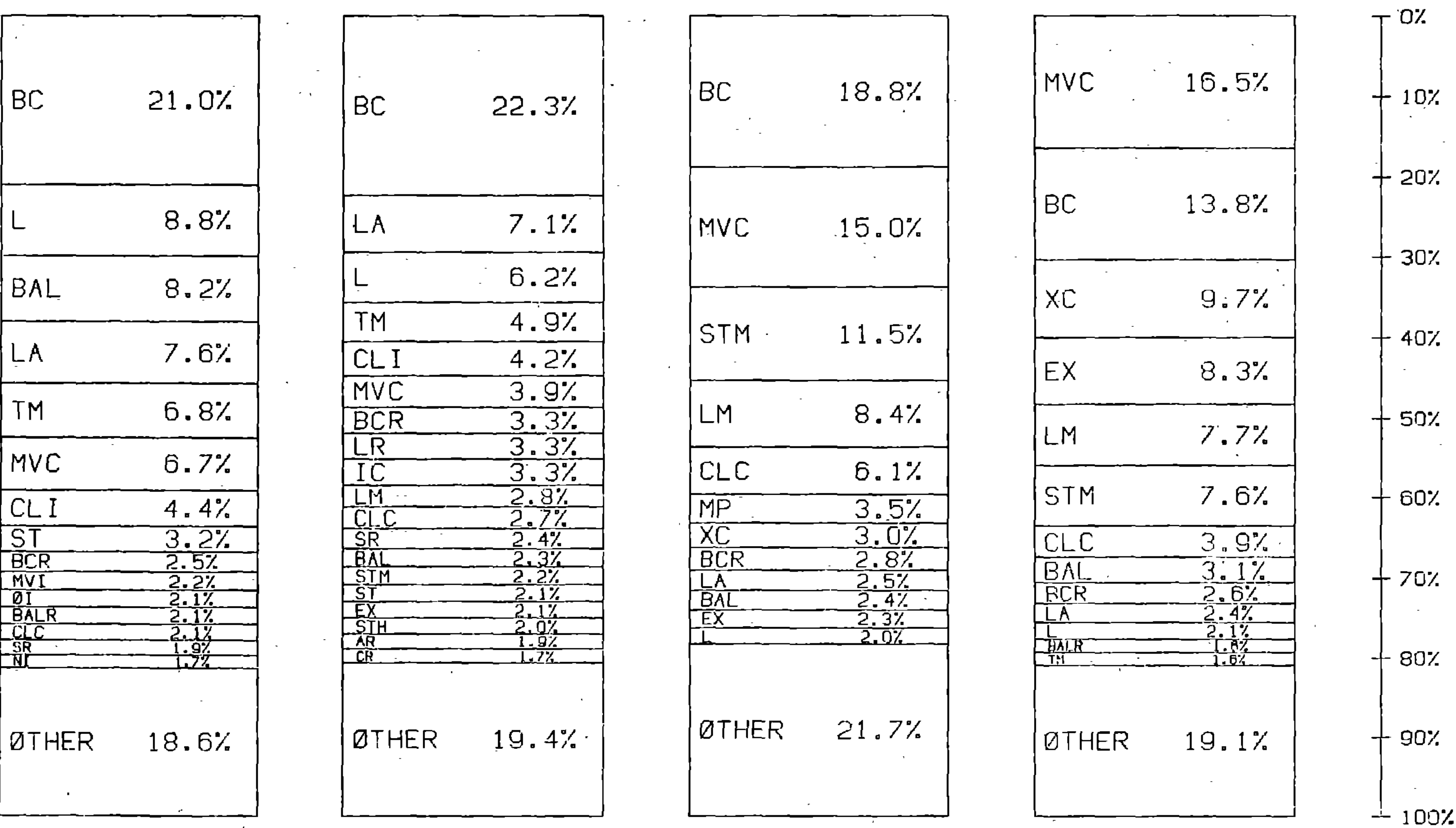


Figure 6: Static/Dynamic Count, and Time by Opcode - COBOLC

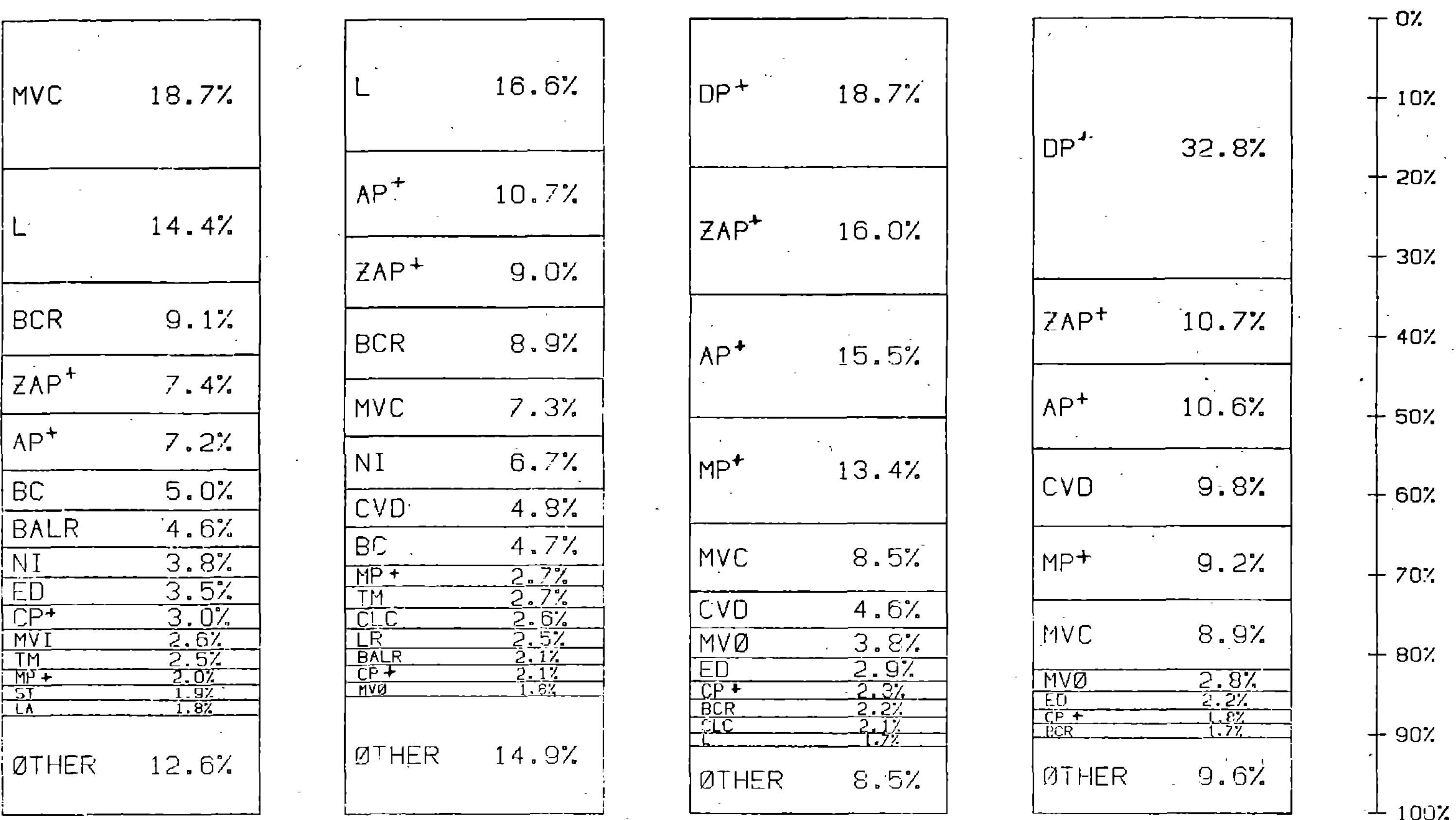
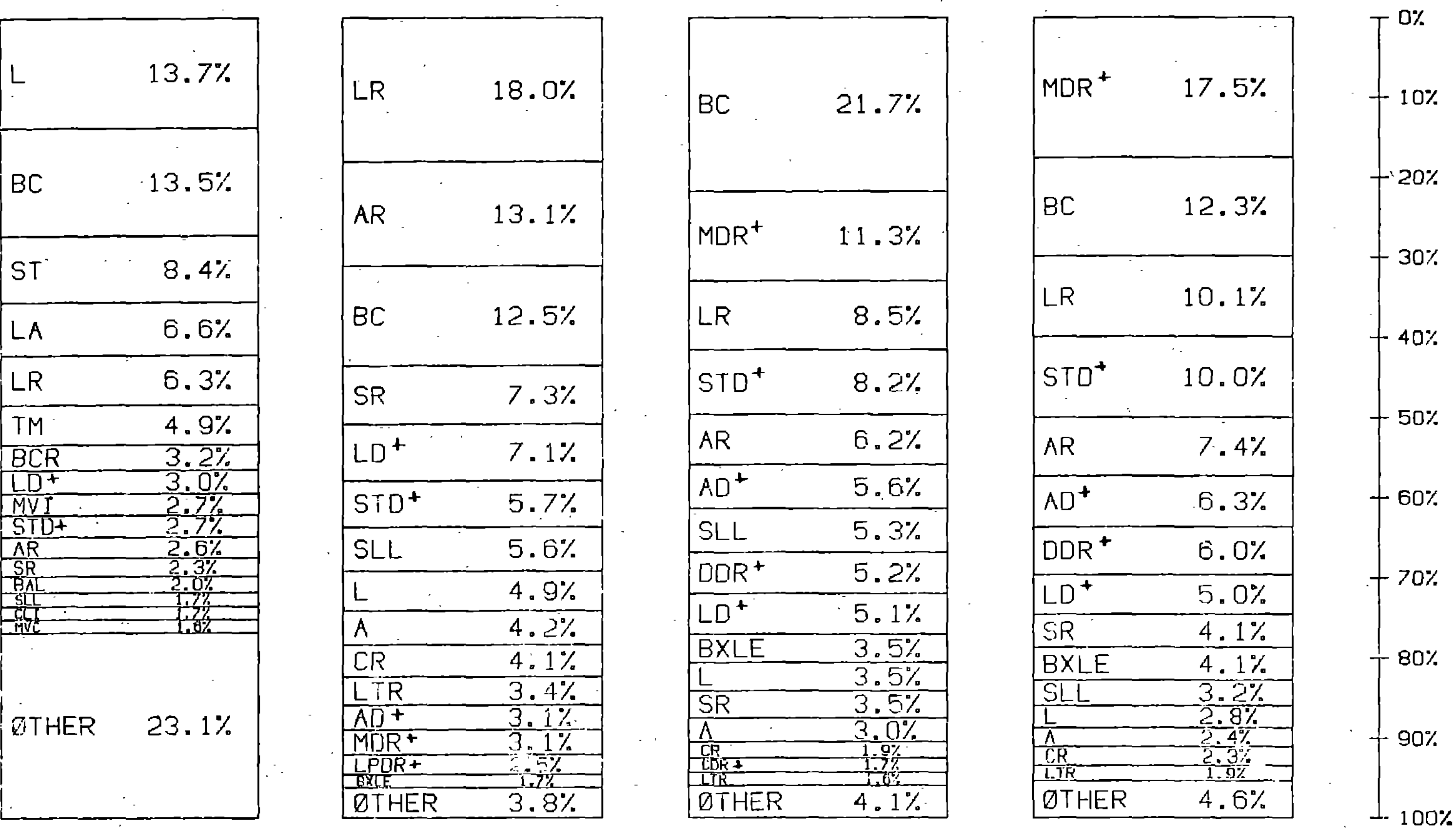


Figure 7: Static/Dynamic Count, and Time by Opcode - COBOLGO



+Floating Point Instruction

Figure 8: Static/Dynamic Count, and Time by Opcode - LINSY2

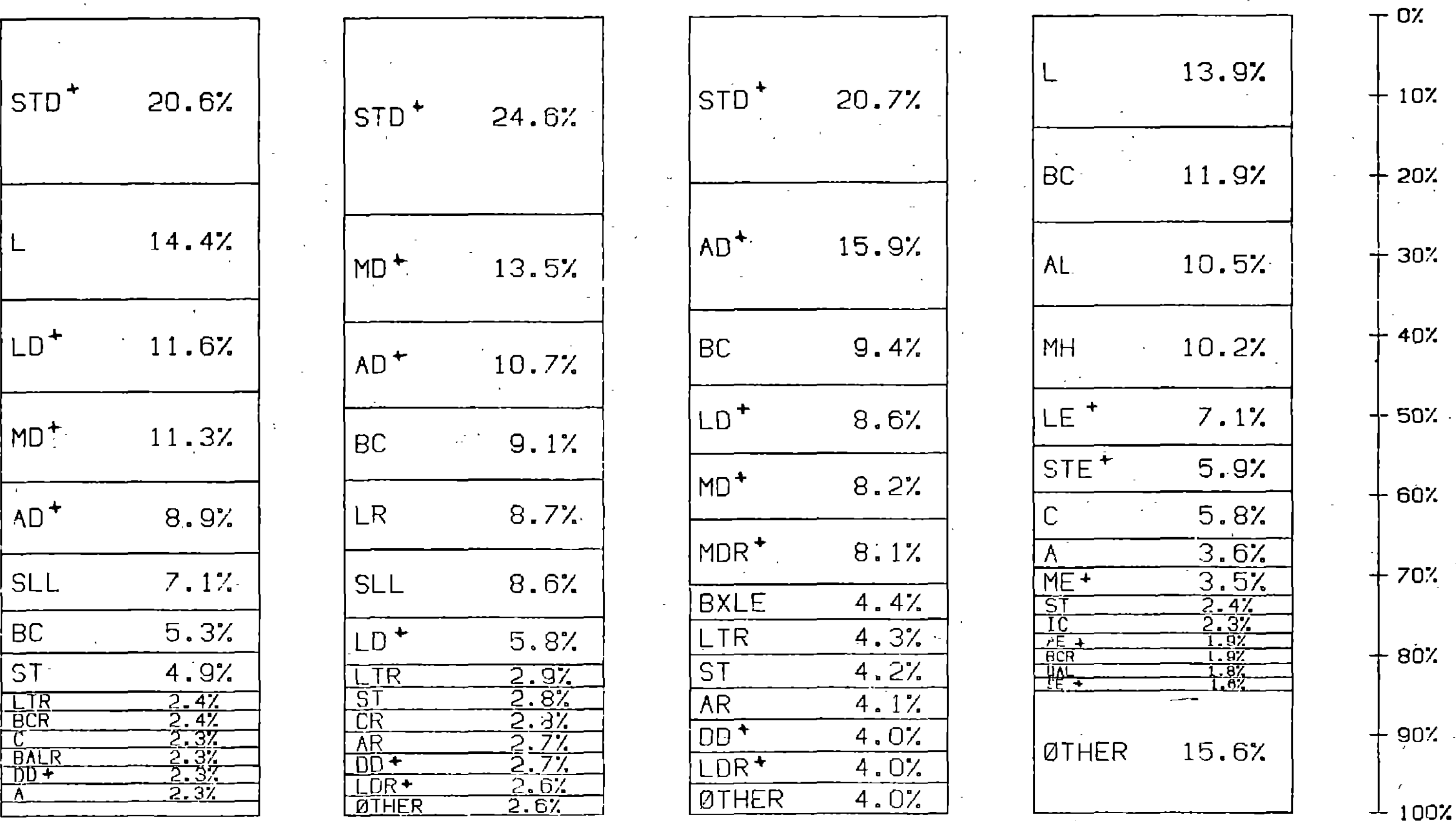


Figure 9: Dynamic Opcode Counts - FFT programs

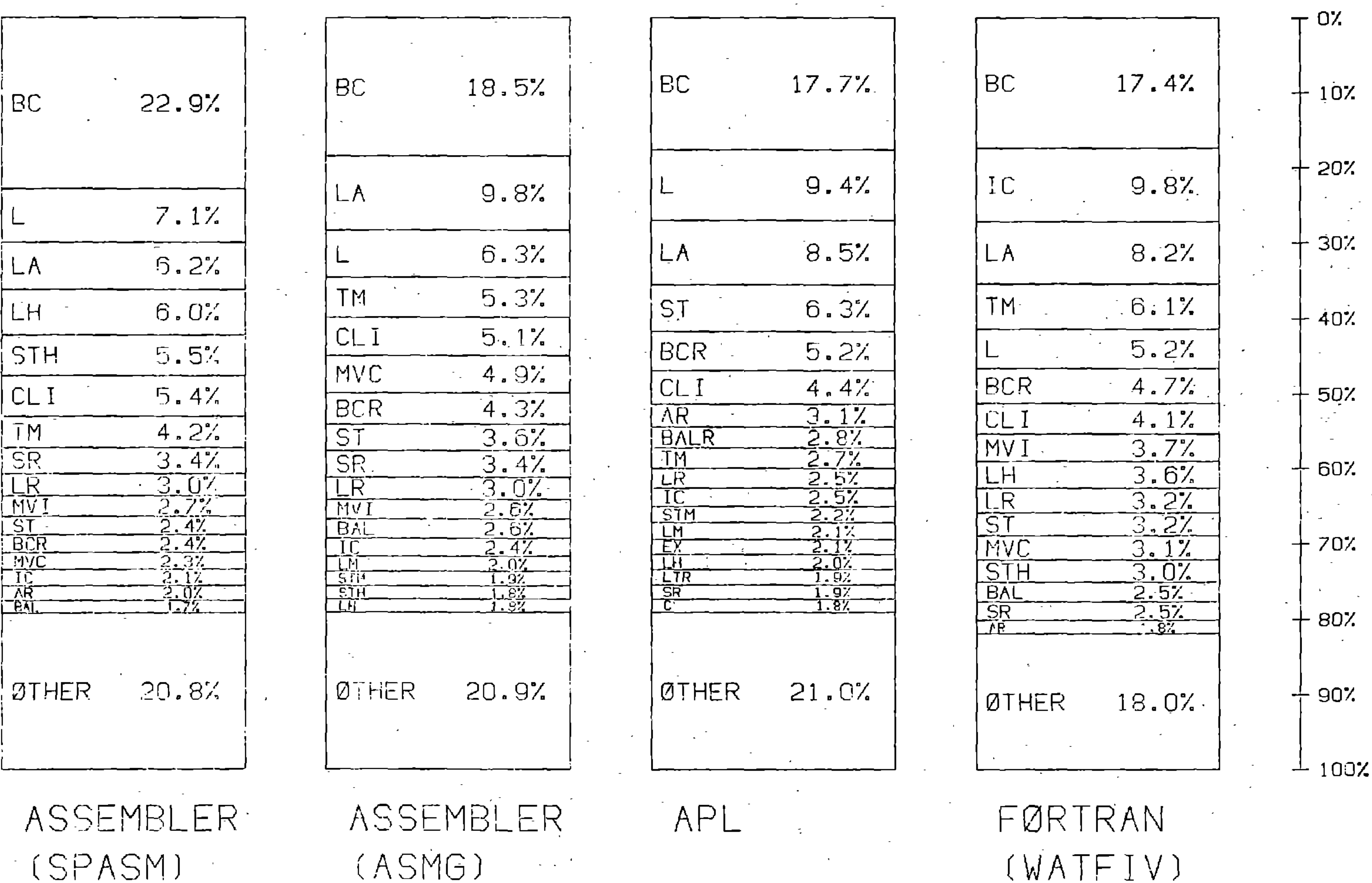
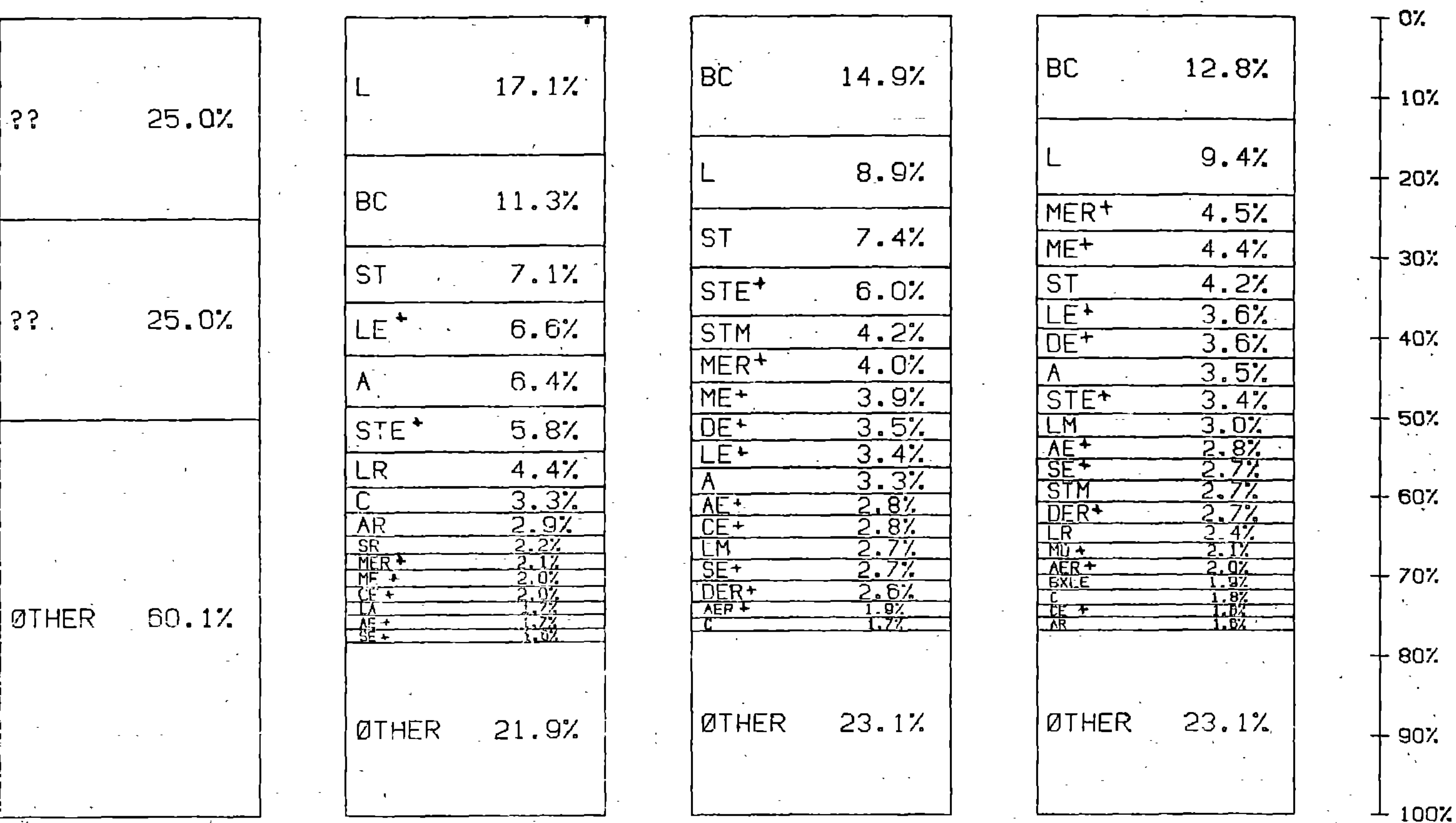


Figure 1Ø: Dynamic Opcode Counts - Language Processors



+Floating Point Instruction

Figure 11: Static/Dynamic Count, and Time by Opcode - LASS

It is clear that the most commonly executed instructions are often not the ones which account for most of the execution time. Some of the more exotic and many of the variable-length instructions of the 370 architecture demonstrate their influence; Divide Decimal (DP) accounts for 18.65% of the Amdahl time for COBOLGO, and Translate and Test (TRT) accounts for 5.38% of the IBM time for PL1C. The particular strengths and weaknesses of the implementations are apparent; the Amdahl implementation of Divide Register (DR) suffers in comparison to IBM (FORTGO), whereas IBM fares rather poorly on Store Multiple (STM). Certain dips in performance are are clearly evident, and two such examples appear in COBOLC. The Execute (EX) instruction, which the Amdahl designers expected not to be important, is a particularly obvious problem, and has been noted before [EME]. The Exclusive Or Character (XC) instruction, which accounts for 8.31% of the execution time, is almost always a case of overlap discussed in section 4.7, which IBM optimized but Amdahl did not.

4.2 INSTRUCTION LENGTH

The 370 architecture has three instruction lengths: 2, 4, and 6 bytes, which loosely correspond to register to register, register to memory, and memory to memory instructions. Table 4 gives the fraction of each type encountered and the average instruction length. The average instruction

length varies from program to program; the range is 2.92 to 4.49, with the average around 3.6 bytes. The extremes are represented by the COBOL programs, for which 6-byte storage to storage instructions predominate, and the LINSY2 program, for which 2-byte register to register instructions predominate. Although the average varies moderately, the proportion of 4-byte instructions varies dramatically from 46% to 81%, and similarly 2-byte instructions vary from 15% to 60%. The high fraction of 2-byte instructions for LINSY2 results from the fact that most of the instructions executed are part of a short (26 byte) inner loop that was highly optimized by the compiler.

Table 4.

Instruction lengths

Program	%2-byte	%4-byte	%6-byte	Average
COBOLC	16.15	75.91	7.94	3.836
FORTGO	29.02	70.69	0.29	3.425
PL1GO	16.99	82.37	0.64	3.673
LINSY2	53.96	46.04	0.00	2.920
COBOLGO	14.74	45.77	39.49	4.495
FORTC	18.52	80.86	0.62	3.642
PL1C	17.20	75.45	7.35	3.803

4.3 BRANCH ANALYSIS

For most programs studied, branch instructions represent a considerable fraction of all instuctions executed (usually 15% to 30%). In five of the seven programs traced, at least one of the branch instructions

41

(usually the simple conditional branch BC) is in the group of instructions that represent 50% of program execution.

In Table 5, the column marked '% Brnchs' indicates the fraction of all instructions executed that were potential branch instructions. The column marked '% Success' which follows, shows the fraction of those potential branches that were successful. In the 370 architecture there are two classes of branches: unconditional branches, and conditional branches whose success depends on values at execution time. Each class contains both successful and unsuccessful branches. The only unusual subclass is the unconditionally unsuccessful branch, which is a no-op instruction. The second part of Table 5 shows the fraction of branches in each of these four subclasses as a fraction of all potential branches encountered.

Table 5.

Branch Instructions

Program	%Brnchs	%Success	Unconditional %Succ	Unconditional %Unsucc	Conditional %Succ	Conditional %Unsucc
COBOLC	31.26	61.75	35.01	6.22	26.74	32.03
FORTGO	13.49	81.81	31.89	6.62	49.92	12.57
PL1GO	6.65	76.04	11.80	9.17	64.25	14.78
LINSY2	14.13	49.34	0.29	0.05	49.64	50.01
COBOLGO	15.78	71.23	35.87	2.75	35.36	26.02
FORTC	21.60	64.41	24.59	3.22	39.82	32.37
PL1C	35.27	67.65	33.50	4.03	34.15	28.32

Branch instructions can create difficulties for pipelined implementations of computer architectures. The instruction fetch mechanism is often a stage in the pipeline which is independent of the instruction decoder, and therefore does not recognize branch instructions. A naive implementation results in a large number of unnecessary instruction fetches following a branch instruction, since the recognition of the need to fetch instuctions from the branch target comes too late.

To address this problem the 168 has a rather sophisticated mechanism by which both the instructions following the potential branch and the instructions at the branch target are fetched into two separate sets of instruction buffers. Although the fraction of success for potential branches seems to be a fairly consistent 60-80%, table 6 demonstrates that it depends heavily on the particular type of branch instruction. The designers of the 168 accounted for this fact by having the instruction fetch mechanism use the specific opcode of the branch to estimate the likelihood of success.

In contrast, the 470 simply treats branch instructions as if they had memory operands, and uses the normal memory operand fetch mechanism to fetch the first two words at the branch target location. Pipeline complexity is minimized by having the execution unit determine the results required for conditional branches as early as possible. This is

Table 6.

Instructions Which Caused Branches, Sorted By Frequency

OPCODE	COUNT	% OF BRANCHES	% SUCCESS FOR THIS OPCODE
BC	1343374	56.365%	60.260% OF 2229306
BCR	555745	23.318%	69.504% OF 799591
BXLE	272120	11.418%	92.208% OF 295116
BALR	97030	4.071%	53.303% OF 182036
BCT	81041	3.400%	96.562% OF 83926
BAL	19646	0.824%	100.000% OF 19646
BXH	14387	0.604%	25.434% OF 56565
BCTR	3	0.000%	0.009% OF 34229
SVC	1	0.000%	0.420% OF 238
	2383347	100.00%	

consistent with the very successful philosophy of the Amdahl designers to keep the pipeline as simple as possible. Since we generally find that branch instructions represent a smaller percentage of the execution time for the 470 than the 168, it appears as though the decision to use a simpler mechanism did not cause a bottleneck.

Section 6.1 contains a proposal, supported by simulation results, for an improved mechanism for branch prediction.

4.4 BRANCH AND EXECUTION DISTANCES

One of the common criticisms of the 370 architecture is the absence of program-counter-relative branch instructions. Figure 12 is a typical branch distance distribution which supports this attack, since 75-85% of the branch distances are within 2048 bytes of the program counter. The

displacement of 12 bits used in RX branch instructions could therefore have been used for most branches so that base registers would have been unnecessary for most program references. The fact that 50-60% of the branch distances are within 128 bytes of the program counter indicates that even an 8-bit displacement could be used to considerable advantage.

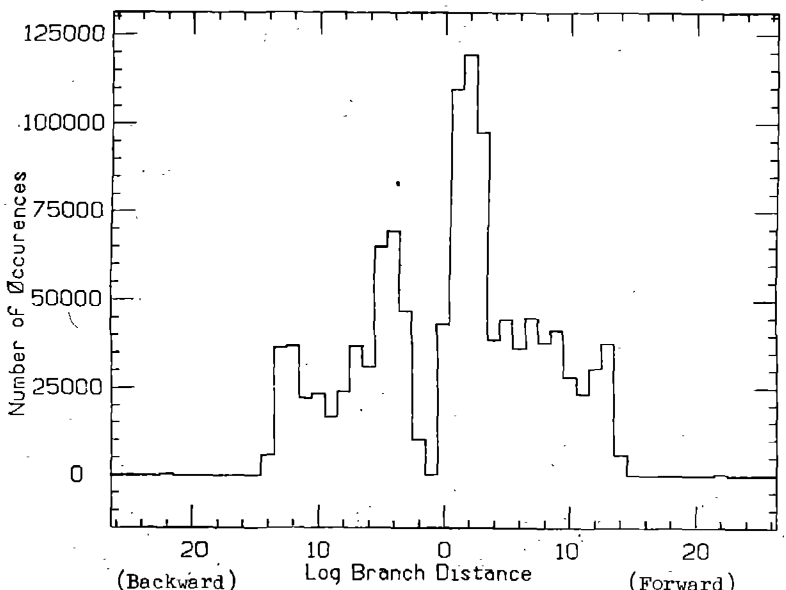


Figure 12: Branch Distances

43

Although 95-99% of the longer branch distances are within 32K bytes, there are still a substantial number of longer branches (4M bytes and above) representing calls to supervisor routines far from the user's program area.

Most programs show a few important peaks in the branch distance distribution corresponding to the important program loops. Note that the asymmetry around the program counter is not sufficient to justify other than a symmetric signed displacement for relative branch instructions.

Table 7 shows information related to execution distances, which is defined to be the number of bytes of instructions executed between successful branch instructions. The last column gives the equivalent distance in number of instructions, obtained by dividing the average execution distance by the average instruction length for that program; it is a reasonable estimate of the true average number of instructions between successful branches.

Table 7.

Execution Distances

Program	Average	Std. Dev.	Avg. # Inst
COBOLC	19.86	17.25	5.18
FORTGO	28.52	31.03	8.33
PL1GO	69.40	34.11	18.89
LINSY2	41.40	25.92	14.17
COBOLGO	33.96	48.07	7.56
FORTC	26.05	25.08	7.15
PL1C	15.94	13.51	4.19

For most programs, the average execution distance is surprisingly small (less than 32 bytes, which is the cache line size) but the standard deviation is large. There are often isolated peaks for relatively large execution distances (see Figure 13). With the exception of the PL1GO program, which has the highest average execution distance, 77% to 85% of execution distances are less than 32 bytes. Distances less than 16 bytes account for 40-60% of the execution distances. This tends to justify the choice of 32 bytes for the linesize of the cache on both machines, at least as far as instruction fetch is concerned. This is also consistent with older designs for instruction fetch buffers, such as the IBM 360/91 which has a 64 byte instruction stack.

4.5 OPCODE PAIR DISTRIBUTION

The measurement of opcode pair frequencies confirms that the overall frequency of an opcode is not independent of the surrounding instructions. Pair occurrences are also important in performance analysis because of pipeline interlocks and other miscellaneous issues such as memory store-through. Table 8 gives the five most frequent opcode pairs for each program. It is not uncommon for the measured frequency of those pairs to be 4 to 9 times greater than the product of the individual opcode frequencies.

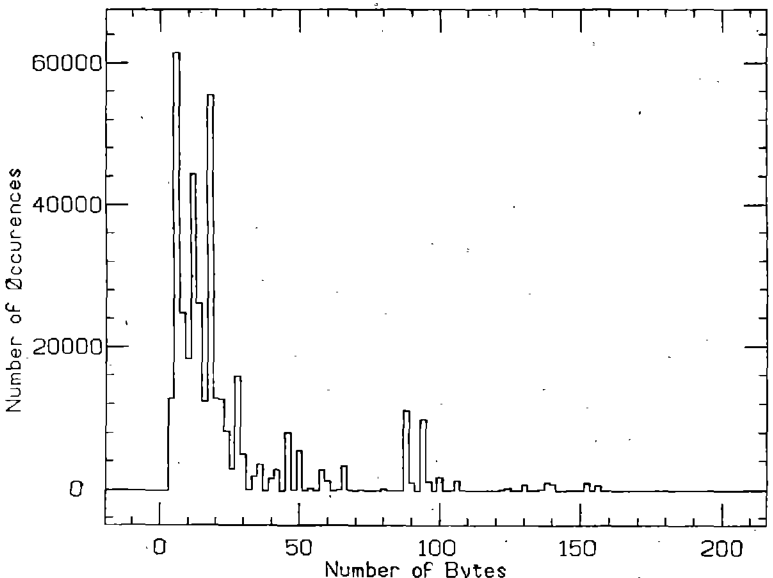


Figure 13: Execution Distance Between Branches

An examination of the frequent opcode pairs fails to discover any pair which occurs frequently enough to suggest creating additional instructions to replace it. Many of the instruction pairs which do occur frequently are those that when combined would save only one opcode field since the other instruction fields would still be required. Examples of this nature are test or compare instructions followed by conditional branches (TM/BC, C/BC). Many other frequent pairs are artifacts of the program structure; a simple

Table 8.

Opcode Pairs

		First Instr	Second Instr	% Pair Count	% Freq. Product	Ratio
COBOLC	1	TM	BC	4.74	1.09	4.36
	2	CLI	BC	4.08	0.93	4.36
	3	CLC	BC	2.67	0.61	4.40
	4	BC	CLI	2.57	0.93	2.75
	5	BC	TM	2.00	1.09	1.84
FORTGO	1	LE	ST	7.37	1.72	6.29
	2	ST	AR	5.34	1.27	4.20
	3	AR	AR	5.29	1.45	3.64
	4	AR	BXLE	5.28	0.64	8.21
	5	BXLE	LE	5.13	0.59	8.66
PL1GO	1	MVI	MVI	7.65	2.51	3.05
	2	AR	AR	7.65	2.20	3.47
	3	AR	L	7.16	4.18	1.71
	4	L	AR	6.67	4.18	1.60
	5	L	A	6.00	1.71	3.50
LINSY2	1	LR	SR	7.26	1.31	5.55
	2	BC	LR	6.65	2.24	2.97
	3	SLL	LD	5.39	0.40	13.54
	4	LR	SLL	5.22	1.01	5.19
	5	LR	AR	4.72	2.35	2.00
COBOLGO	1	L	BCR	5.79	1.48	3.92
	2	AP	NI	5.20	0.72	7.28
	3	L	CVD	4.21	0.79	5.31
	4	NI	L	3.96	1.11	3.58
	5	BCR	L	3.73	1.48	2.52
FORTC	1	BC	L	6.29	3.57	1.76
	2	L	L	6.19	7.54	0.82
	3	ST	L	4.03	3.34	1.21
	4	L	BCR	3.76	1.28	2.94
	5	L	ST	3.66	3.34	1.09

PL1C	First Instr	Second Instr	% Pair Count	% Freq. Product	Ratio
1	CLI	BC	6.54	1.65	3.96
2	BC	LA	4.20	1.90	2.22
3	BC	CLI	3.76	1.65	2.28
4	TM	BC	2.93	0.79	3.71
5	CR	BC	2.26	0.58	3.89

example is the pair which consists of a loop branch and its target instruction. Alexander [ALE75] mentions the load-branch pair as an extremly frequent one for the XPL compiler (L-BC is 12.4% of the count) where it is used for the GO TO in the absence of a known base register. We find no pairs with such high frequencies, and in particular find the load-branch combination to be significant only in two of the seven programs. Frequent pairs often result from peculiarities of software conventions; the subroutine-call instruction (BALR) is often followed by the unconditional branch (BC) because the first instruction in almost all subroutines is a branch around the name of the program. For the FORTGO program, the extra branches (which could be easily eliminated by putting the name before the first instruction of the subroutine) cost 0.70% of the execution time of the entire program. Many of the programs have a similar extra cost of between 0.5% and 1.0% due to the same convention.

The distinction between the distribution of instruction pairs executed and the static distibution of instruction pairs in the program text should be carefully made. Our

results do not contradict findings based on static analysis [FOS71a, HEH74] that certain pairs of instructions might be frequent enough to justify replacement by a single instruction to improve code density.

4.6 DISPLACEMENT VALUES

The 370 architecture expresses addresses as the sum of a 24 bit base value in a register with a 12 bit displacement in the instruction. The limited size of the displacement -- which is interpreted as an unsigned number from 0 to 4095 -- is the source of much of the criticism of the instruction set. The small displacements force many registers to be dedicated as program base pointers, and make local variable accesses awkward. Figure 14, which shows the log distribution of displacement values as measured statically in the program, supports this allegation. Each additional bit which makes the displacement larger is used about as much as the preceeding bits; it is not the case that small displacements, like small data constants, are common. It is likely (as supported by the evidence) and obvious (to assembly language programmers) that a larger displacement would be a considerable advantage.

4.7 REGISTER USE

All memory addresses in the 370 are formed with a 24 bit base value in a register which is added to the

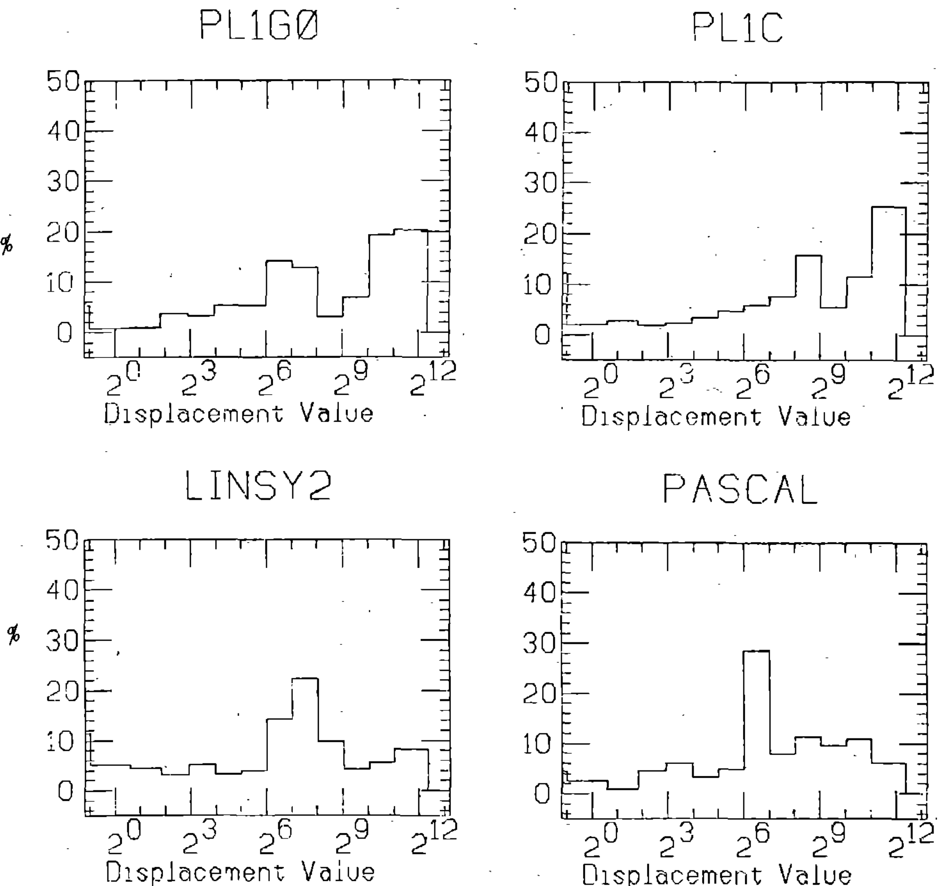


Figure 14: Distribution of Displacement Values

displacement, and some instructions allow an additional 24 bit quantity in a second register to be used as an index. In all cases specification of register Ø for the base or index indicates that a value of zero is to be used in lieu of the contents of the register. The hardware does not distinguish between registers which contain addresses and registers which contain index values, so the interpretation of statistics about base and index register utilization are difficult to relate to the program organization. Nevertheless information about the occurrence of zero in the register fields can be easily interpreted. Table 9 shows that it is very infrequent for instructions to specify the use of both index and base registers. Except for the program LINSY2, which is known to have many array references, 8Ø% to 95% of the instructions which allow indexing do not use both base and index registers. A reorganization of the 37Ø addressing modes could profitably include a non-indexed mode in which the space saved is used for a longer displacement.

The distribution of register utilization for address calculation shows that no more than 3 registers account for most of the use. The others are used for address calculation less frequently, or are used for program accumulators.

Table 9.

Register Use for RX Instructions

(Effective Address Calculation)

Program	%No Regs	%1 Reg	%2 Reg
COBOLC	0.39	95.51	4.09
FORTGO	0.96	77.25	21.79
PL1GO	0.09	82.05	17.86
LINSY2	0.24	65.04	34.72
COBOLGO	0.01	98.93	1.06
FORTC	4.08	87.95	7.97
PL1C	1.93	92.48	5.59

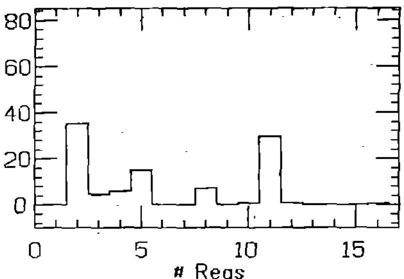
4.8 OPERAND LENGTHS

The TRACE program accumulates the distribution of the lengths of all the operands for instructions for which the operand lengths are not implied by the opcode. These operand lengths are either fixed and defined in other fields of the instructions (like the number of registers specified in the Load Multiple instruction), or are data dependent (like the number of bytes which must be referenced before an inequality is detected in a Compare Character instruction). These variables are required to calculate the instruction execution times.

For the purposes of exposition we have divided the variable operand length instructions into three classes: (1) the multiple register load and store instructions (LM and STM), (2) the character manipulation instructions, like Move Character (MVC), and Compare Character (CLC), and (3) the decimal arithmetic instructions like Add Decimal (AP).

4.8.1 LM/STM

The STM and LM instructions save and load a contiguous set of registers designated by a starting and ending register. From one to sixteen registers may be moved by a single instruction. Figure 15 shows a typical distribution (from FORTGO) of the number of registers stored and loaded. It is common for there to be two peaks, one for a low value of about 2 to 3 registers for accessing data stored in consecutive words, and another at a high value of 11 to 15 registers for saving and restoring registers across procedure calls. The LM and STM are not used symmetrically: for a given number of registers loaded or stored the frequency counts are often quite different. For the FORTGO program, the average number of registers used for STM is 13.23, and for LM is 5.99. For both machines, the marginal cost of storing one more register is smaller than the execution time of a load or store instruction, but there is a much higher overhead for starting each instruction for IBM than for Amdahl. In both cases it is faster to use several store or load instructions when 3 or fewer registers are involved. Despite the fact that these instructions are never among the most frequent, they contribute much more to the CPU time than their frequency would suggest because of their long execution time. For the FORTGO program for example, the 0.67% of instructions which are STM account for 6.66% of the IBM execution time and 4.59% of the Amdahl execution time.



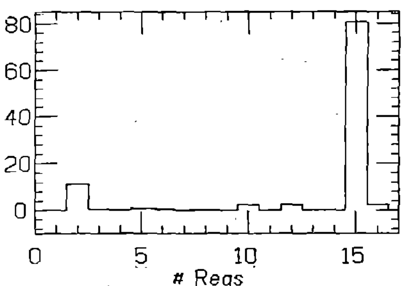


Figure 15: Number of Registers for LM/STM

4.8.2 Character Instructions

The second group of storage-to-storage (SS) instructions are those which specify a source and destination location for a character string and a single length for both operands in the range 1 to 256. One of the characteristics of these instructions that makes their implementation very difficult is that overlapped operands are allowed and must be treated a byte at a time. This allows, for example, a single byte to be propagated throughout a string by a move

instruction whose destination address is one greater than the source address, since the fields are processed left to right. Lower performance machines in the 370 family implement these instructions in all cases by processing each byte individually, but for high performance machines this would obviously be too slow. Therefore both computers exhibit execution speeds for the non-overlapped cases which are much higher than that for overlapped. For the IBM Move Character instruction, for example, the non-overlapped case takes 40 nsec per byte moved, but 240 nsec per byte of overlapped move.

On jobs for which MVC is a frequent instruction (PL1C and COBOLC) the nonoverlapped case occurs about 50 times more frequently than the overlapped case. However the average number of bytes moved is less than 8 for the nonoverlapped move, and greater than 50 for the overlapped move. The result is that the 2% of the MVCs which are overlapped are responsible for 20% of the total MVC time.

The overlapped MVC instructions are used primarily to fill a work area with a specific character, and are probably most used to initialize I/O buffers. This is confirmed by the peaks near 80 and 133 which correspond to card and line printer buffers. For programs which don't otherwise use MVC but still do I/O, the overlapped case is an even higher fraction of all occurrences of MVC. For FORTC, for example, the 6% overlapped MVCs account for 52% of the MVC time.

Figure 16 shows the distribution of operand length for MVC instruction in FORTC. It is representative of the other distributions in the presence of large peaks for small values, and an overall average of 10.06 bytes. Since the startup overhead for these instructions is large, there is almost always a less expensive way to do the equivalent operation for a small number of bytes. For one byte, a IC/STC combination takes less than half the time of a one-byte MVC on both machines.

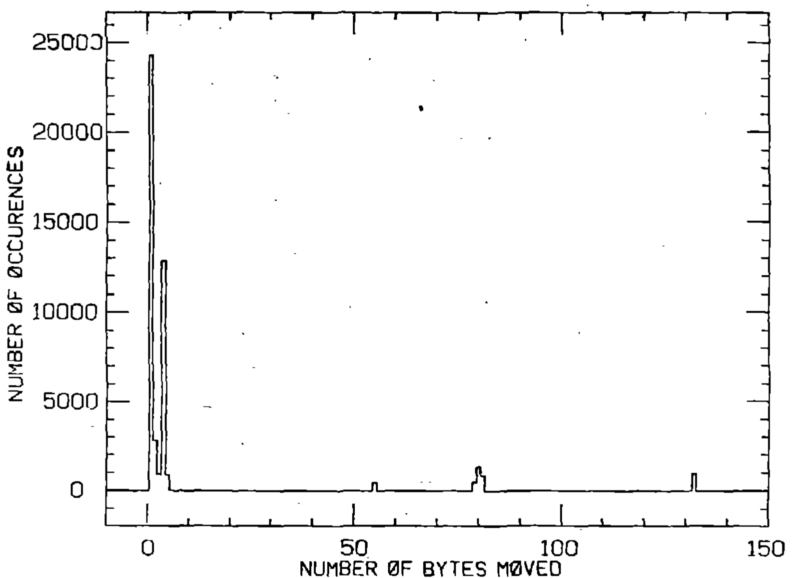


Figure 16: MVC operand length

Most of the other instructions in this variable operand class are much less frequent than MVC. Among them are the instructions for which the number of bytes processed may be much smaller than indicated in the instruction, such as Compare Character (CLC) and Translate and Test (TRT). For these instructions, the distribution of the length specified in the instructions is a poor indicator of the length actually used. A typical examples is COBOLC, where the average CLC instruction specifies 4.53 bytes, but an average of only 1.74 bytes are examined by the hardware.

Another instruction of note is the Exclusive Or Character (XC) which is predominately used in total overlap mode in order to zero fields. This fact was used to advantage in the 168, where the total overlap case is specially optimized to be 15 times faster than the other overlap cases. This was not done for the 470, which explains that XC accounts for 9.6% of the COBOLC program for the 470, but only 3.0% for the 168.

4.8.3 Decimal Instructions

The third group of storage-to-storage instructions consist primarily of those for decimal arithmetic. They appear in significant numbers only in the COBOLGO program. For that program, however, they account for 26.29% of the count, and represent 66.39% of the IBM execution time and 64.30% of the Amdahl execution time. These instructions can

vary in execution time by as much as 16 to 1 depending on the operand lengths, but the large execution time arises despite the fact that relatively short operands are common. Most operands are 2 to 6 bytes long even though the maximum possible is 16. The average execution time of the Divide Decimal (DP) instruction is about 15 usec for both machines. Not suprisingly, the average execution rate in millions of instructions per second for the COBOLGO program (.810 MIPS for IBM, 1.353 MIPS for Amdahl) is drastically smaller than the average for all the programs (3.519 MIPS for IBM, 5.518 MIPS for Amdahl). Considering the popularity of COBOL as a programming language, these instructions, which require slow serial byte processing, represent a major degradation of the speed of the machines.

In view of the poor performance of many of the variable operand length instructions, their inclusion in the the architecture of a high-performance computer is questionable. The absence of such instructions in machines like the CDC 7600 and the CRAY-1 is indicative of their emphasis on high speed. The arithmetic which must occur before these instructions begin their data transfer suggests that it is quite difficult to optimize them for short operands. A compromise, if the execution of these instructions cannot be optimized, may be to supply simpler instructions from which the more complex character and decimal instructions can be composed, as illustrated by the byte instructions of the

PDP-10. An immediate improvement could be obtained if compilers were to replace these instructions by faster equivalents when they are available, although this would require tailoring the compilers to specific models of the computer series.

4.9 MEMORY REFERENCE AND CACHE EFFECTS

The correction due to cache misses ranges from 1% to 5% for IBM, but from 3% to 19% for Amdahl, indicating that the memory subsystem is a major bottleneck for the Amdahl machine. In some sense the memory architecture forces the 470 to lose some of the raw speed advantage of the CPU. There are two factors which contribute to the problem: (1) the cache organization of the Amdahl machine produces from 1.7 to 3 times the number of cache misses, and (2) the penalty for each miss is 1.56 times that for IBM. Thus the overall cache penalty for Amdahl is 2.5 to 4 times more than IBM, whereas the raw execution speed, defined as Tins (the time required to execute the instructions with no cache misses) is 1.9 times faster for Amdahl than for IBM. The loss due to the cache organization could have been eliminated, but to maintain the raw speed advantage would have required a cache miss penalty of 250 nsec, which would not have been economically feasible at the time. The dilemma of Amdahl results from a mismatch between the MOS memory chips available commercially and its proprietary ECL LSI technology which is so much faster.

4.10 PIPELINE EFFECTS FOR THE 470

Because the timing formulas for the Amdahl machine include specific pipeline variables, we can assess their effect on the execution. The pipeline is optimized for 4-byte instructions which have single word operands, and any deviation causes potential conflicts with subsequent instructions.

The seven pipeline variables depend upon local instruction sequences (see the definition of S1 in section 3.1.4 for examples), and therefore cannot be computed from global averages. The exact evaluation of these variables would require a complete and complex simulation of the pipeline at the time the program is traced. As a compromise, we use the pair and triple frequency data collected while tracing to reconstruct instruction sequences and average the variable value for each sequence.

In general, the speed degradation due to pipeline conflicts seems to be quite small. For most programs, each of the variables contributes less than 0.5% to the total execution time. The only cases of a larger contribution are when the variables affect specific instructions which occur frequently. For the COBOLGO job, an average additional 1.1 cycles (35.75 nsec) is added to each decimal instruction. This represents a 1.35% increase in execution time. For PL1GO, the doubleword store pipeline break results in an

additional 1.17%. For LINSY2, the delay caused by late setting of the condition code needed for conditional branches adds 0.3%. Although there are wide variations, these worst case examples demonstrate the overall good design of the pipeline.

4.11 SUMMARY

Results of two types are evident in the measurements presented here: specific statements about the 168 or 470 implementations, and general observations about the 370 architecture. This is precisely the advantage of doing a study at this level, because the effect of architecture on implementation is made clear in a way that is hard to see otherwise.

Comparing the two implementations leads to the general impression that the 168 is a more balanced design. All of the subsystems seem well matched to each other, and it is hard to identify particularly vulnerable "critical paths" to improved performance. The 470, on the other hand, benefits from improved technology but suffers from uneven improvement compared to the 168. There are particular isolated items - certain special instructions and the cache/memory organization, for example -- which have not scaled up in proportion with the rest of the machine. The result is that specific attention paid to those areas can have a significant effect; in essence the design seems to need another

pass before the result is performance which seems "consistent" among all its parts.

An improved version of the 47Ø V/6 has recently been announced, and some improvements to the areas discussed here have been made; in particular the cache has been improved, the the XC instruction has been optimized for the fully overlapped case, and the EX instruction no longer causes a complete 22 cycle pipeline break. Not to be outdone, an improved version of the 168 (labeled the 168-3) has been produced. In addition to changes unrelated to performance (the addition of an independent diagnostic computer, for example) there was improvement in a few of the important storage-to-storage instructions and a doubling of the cache to 32K, resulting in a speed increase of several percent.

The overall impression of the architectural measurements is two-fold: (1) the basic instruction set seems balanced and complete[2], and (2) there is a significant problem with high-performance implementations of complex instructions, and their use may not be justified. More discussion of this issue appears in Chapter 7.

---

[2] This impression will be somewhat modified by the results of Chapter 5.

Chapter V

OTHER ARCHITECTURE CHARACTERISTICS

This chapter contains an examination of some architectures which are different from the IBM 37Ø. The intent is to complement the 37Ø analysis by examining those aspects of several other computers that can contribute new information because of their difference from the 37Ø.

The architectures for which information is given are: the INTEL 8Ø8Ø microprocessor, the DEC PDP11, and the PASCAL PCODE pseudo-machine.

5.1 INTEL 8Ø8Ø

The LSI microprocessor has sparked a minor revolution in the use and distribution of computing ability. These relatively sophisticated but inexpensive components have permitted both the application of standard computer techniques to a wider body of problems and the development of new applications to problems previously approached in other ways. It is natural to apply the same measurement techniques for this extreme of the computer spectrum that we have applied to examples of the largest computers, and in this way attempt to determine if any fundamental differences

53

in their use can be measured which arise from differences in architecture or application.

The microprocessor used for this study is the INTEL 8080 [INT], which is the oldest and most widely used of the "second generation" single-chip microprocessors. An 8080 simulator running on a large computer was instrumented so that information about instruction execution could be collected, and additional programs were written to analyze source programs so that static information could be obtained.

Gathering benchmark programs for microprocessors is more difficult than for large computers because many of the programs so far written are trivial and uninteresting. Five large programs were analyzed, none of which was written especially for this study. JBASIC is a standard 8K BASIC interpreter which also includes a simple program editor. TBASIC is a much smaller BASIC interpreter which is implemented as a two-level layered machine [WAN]. VGT is the software used in a sophisticated text and graphics terminal [SHU]. MUSIC is a realtime music synthesis and display pattern generator, and EDITOR is a disk-oriented program and text editor. The text editor was written in the high-level langauge PL/M [KIL]; all the others were written directly in assembly language for the 8080.

Table 10.

8080 Dynamic Instruction Statistics

Program	#Inst	-----Instruction Size------ % 1-byte	% 2-byte	% 3-byte	Avg. bytes per Inst
VGT	85,798	52.50	11.24	36.26	1.838
TBASIC	177,486	64.90	13.37	21.73	1.568
MUSIC	168,768	70.30	19.63	10.07	1.398
JBASIC	226,826	76.61	4.03	19.36	1.428
EDITOR	197,330	73.51	10.57	15.92	1.424

Program	Bytes Read per Inst	Bytes Written per Inst	# Cyles per Inst	MIPS at 2 Mhz
VGT	.350	.303	8.967	.223
TBASIC	.343	.204	7.851	.255
MUSIC	.178	.165	6.771	.295
JBASIC	.226	.169	7.002	.286
EDITOR	.209	.160	7.024	.285

5.1.1 Opcode Distribution

As expected, a small number of the 8080 opcodes account for most of each of the program's execution. In the VGT (Figure 17) only seven instructions represent 53.7% of all instructions executed, and 39 represent 99.2%. The most common instruction (14.9%) is the conditional jump (JMP CC,xxx) followed closely by the 8-bit register-to-register load (LOD r,r). In the EDITOR, the same two instructions are important, but the more unusual rotate instruction (ROT) appears in second place because it it used in the inner loop of the multiply subroutine. For the VGT the rotate appears only as the 34th instruction; clearly some generally infrequent instructions are occassionally very important for

specific programs. The jump, load, push (PUSH rr) and pop (POP rr) instructions, however, appear to be universally important for all programs.

There is a considerable difference between the occurence distribution and the time distribution of opcodes for the 8080, but not nearly as much as for the high-performance 370 machines. Instructions like the unconditional subroutine call (CALL U,xxx) are more significant in execution time than in frequency (7.2% vs. 3.8% for VGT) because of the costly stack references to memory, but simple instructions like LOD take less time than their frequency would indicate (7.5% vs. 11.4%).

The static opcode distributions (Figure 18) are often quite different from the dynamic distributions. Although loads and jumps still predominate, lengthy but infrequently executed initialization code is represented by the presence of the load-immediate instructions (LODI r,n for 8-bit data, and LODI rr,xxx for 16-bit data) in the top 50% group. The simple byte movement instructions are statically common, but the dynamically important stack push and pop instructions are not.

5.1.2 Opcode Pairs

Dynamic opcode pair frequencies often clearly reveal the dominant loop of an executing program. In TBASIC for example (Table 11), a string search constructed from an

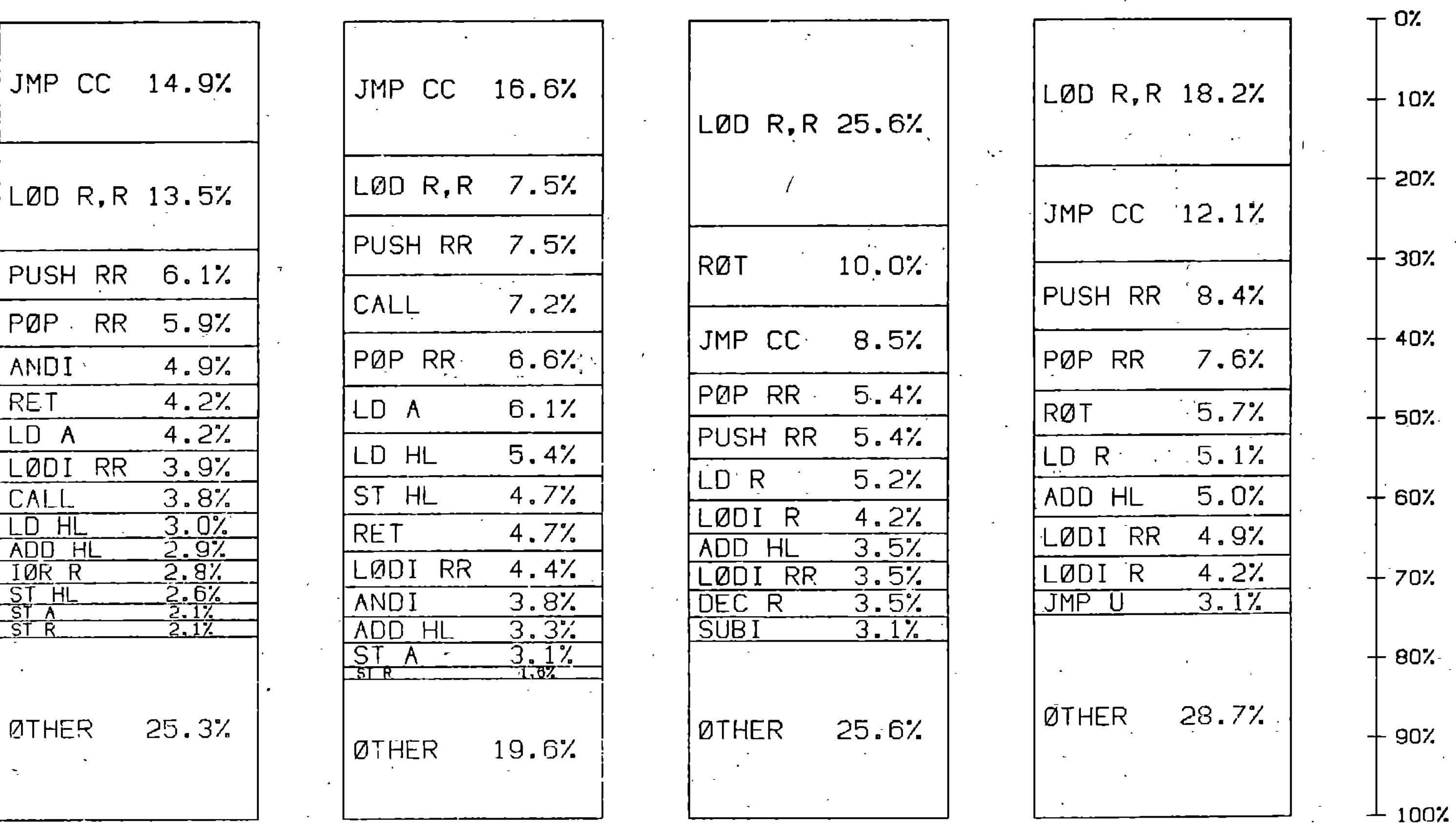


Figure 17: Intel 8080 Dynamic Instruction Count, and Time by Opcode

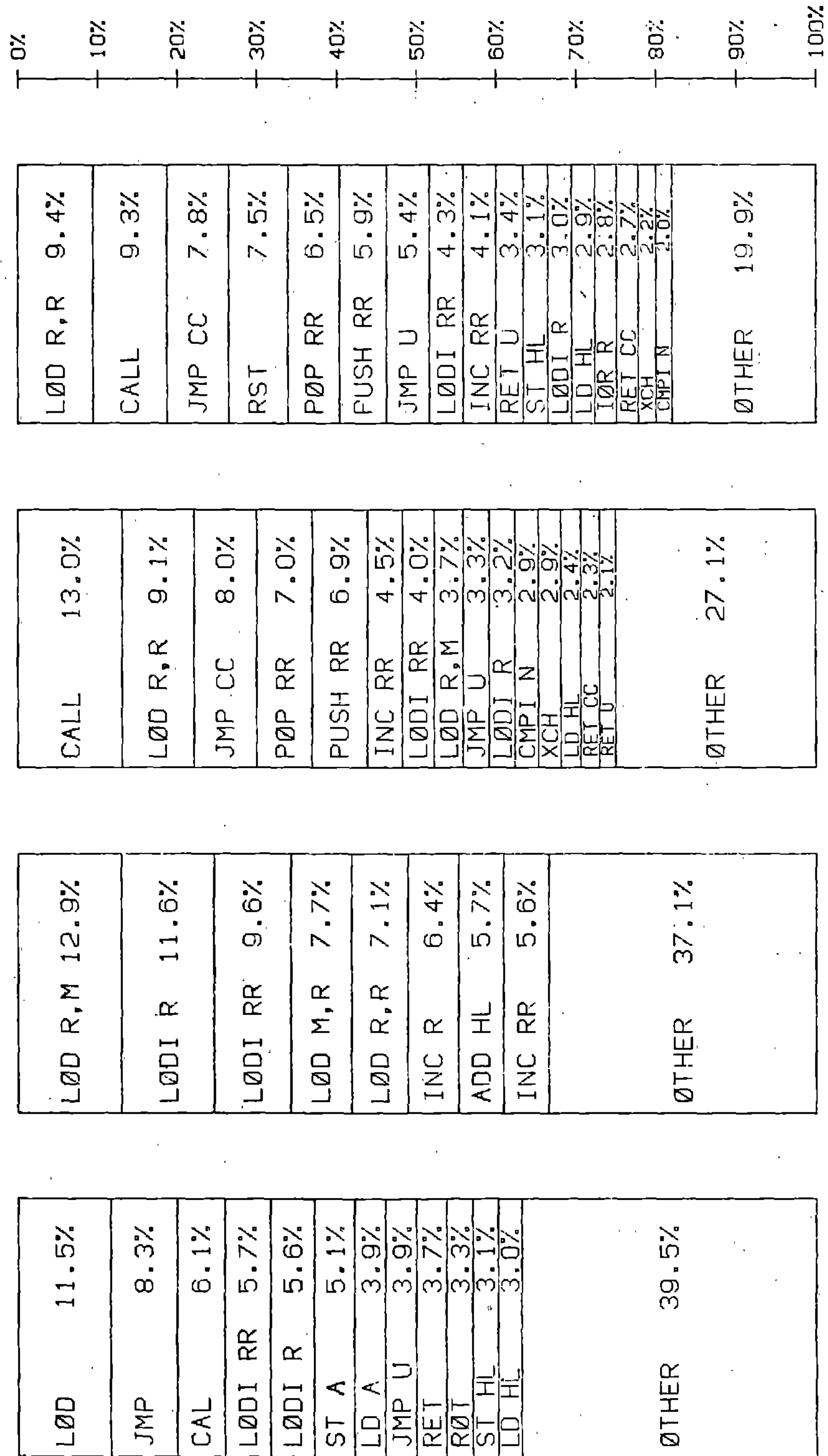


Figure 18: Intel 8Ø8Ø Static Opcode Distribution

index register increment (INC HL), a character comparison (CMP (HL)), and a conditional jump (JMP CC,xxx) represents 9.3% of the program execution, and a slightly longer variation ending with the same sequence represents an addition 4.9%. These simple sequences that could well be replaced by a single faster instruction (such as the compare, increment and repeat (CPIR) of the Zilog Z80) are seldom seen in the more complete 370 instruction set statistics.

Table 11.

8080 Dynamic Opcode Pairs

Program: VGT

Opcode #1	Opcode #1	% Inst Measured	% Inst Expected	Ratio Meas./Exp.
PUSH RR	PUSH RR	3.72	0.38	9.89
ANDI n	JMP CC,xxx	3.02	0.72	4.16
JMP CC,xxx	LOD R,R	2.84	2.02	1.41
LOD R,R	ANDI n	2.84	0.66	4.31
POP RR	POP RR	2.40	0.35	6.85

Program: TBASIC

Opcode #1	Opcode #1	% Inst Measured	% Inst Expected	Ratio Meas./Exp.
JMP CC,xxx	INC RR	6.69	1.71	3.91
CMP (HL)	JMP CC,xxx	6.07	0.96	6.30
INC RR	CMP (HL)	4.76	0.82	5.83
CMPI n	RET CC	3.63	0.39	9.36
CMPI n	JMP CC,xxx	3.47	1.03	3.35

The static opcode pair frequencies (Table 12) often reflect common code sequences. Some of these sequences, like a compare (CMPI n) or a test (IOR r) followed by a conditional jump, are reasonable and unsurprising. Many others are indications of common sequences that should, from an architectural point of view, be incorporated into a single instruction. Rotate instructions often followed by other rotate instructions show that multiple-bit rotation is indeed a common operation; note that the rotate pair is 12 times more frequent than would have been expected from a simple count of its occurrences. Pairs of 8-bit register-to-register loads are common because they are used to simulate the missing 16-bit register-to-register loads. The static pair distribution for the EDITOR reflects the simple code-generation schemes used by the PL/M compiler. Addressing is often done by loading an address into the HL register and referencing the variable in a subsequent register-indirect instruction since the available full-address instructions are limited.

5.1.3 Instruction Length

The 8080 has instructions ranging from a single byte to three bytes. The first byte is always the opcode, which includes register designators, and the following bytes are either immediate data or memory addresses. The average instruction length for the samples varies from 1.4 to 1.8 bytes.

Table 12.

8080 Static Opcode Pairs

Program: VGT

Opcode #1	Opcode #2	% Inst Measured	% Inst Expected	Ratio Meas./Exp.
LOD R,R	LOD R,R	2.59	1.33	1.95
CMPI n	JMP CC,xxx	1.74	0.18	9.71
ROT	ROT	1.69	0.14	12.26
IOR R	JMP CC	1.22	0.10	11.89
LODI R,n	ST A,xxx	1.08	0.28	3.83

Program: EDITOR

Opcode #1	Opcode #2	% Inst Measured	% Inst Expected	Ratio Meas./Exp.
ST R,(HL)	INC RR	3.29	0.43	7.73
INC RR	ST R,(HL)	3.17	0.43	7.41
LODI RR,xxx	LOD R,(HL)	3.10	1.24	2.49
INC R	LODI R,n	3.05	0.74	4.09
LOD R,(HL)	INC R	2.89	0.72	4.00

5.1.4 Branch and Execution Distance

The 8080 also suffers from the absence of jump instructions which are relative to the program counter; as indicated in Table 13, about 80% of the successful branches are made to locations within 127 bytes of the jump instruction. Considering the importance of jump instructions, a sizeable savings in program size as well as speed (because the jump instruction can be smaller) results from the introduction of a relative jump.

Table 13.

Types of 8080 Jumps

Progm	Jumps as % Inst	---- Types of Jumps ---- %Uncond	conditional % Succ	% Unsucc	Branch Distances --- All Branches -- 0 to 127	0 to -127
VGT	16.15	7.62	58.08	34.30	65.36	27.69
TBASIC	17.64	19.43	33.55	47.02	25.13	64.83
MUSIC	4.67	26.72	38.92	34.36	37.58	56.59
JBASIC	8.94	13.63	41.81	44.55	36.75	27.95
EDITOR	10.65	20.45	51.21	28.34	47.61	43.83

The number of instructions executed between successful branches - the execution distance - is just as low for the microprocessors as for the larger machines. Table 14 shows that typically only 5 to 10 instructions on the average are executed in a single run. Although this is not as important for microprocessors because the execution pipeline is small or non-existent, it is surprising that the relatively unsophisticated instruction set does not lead to longer code sequences for most operations. Part of the explanation is that the simple operations to which all these computers devote most of their time are single instructions in both machines, such as the load/store/test operations. The complex operations which would result in long code sequences for the 8080 may not appear simply because either they contain conditional subparts (which break the execution run), or because the relative simplicity of the applications does not require such operations. There is also a feedback effect: the lack of some operations results in the choice

of a fundamentally different algorithm, rather than the original algorithm with the missing instructions simulated. The set of available primitives often strongly affects, as well it should, the design of efficient algorithms, programs, and systems; this is a point that the devotees of rigid top-down structured programming techniques do not sufficiently address.

Table 14.

8080 Execution Distance

Program	Average (bytes)	Std. Dev.	Average (instr)
VGT	9.526	6.771	5.184
TBASIC	7.098	6.629	4.526
MUSIC	10.547	11.770	7.546
JBASIC	12.685	10.287	8.886
EDITOR	15.879	13.888	11.150

5.1.5 Memory References

The number of data references per instruction is important because the data references contribute directly to the exection time; there is little overlap of memory accesses with instruction execution. It turns out, however, that compared to the cost of instruction fetch, operand accesses are a minor part of the execution time. Table 10 shows that the number of operand bytes read per instruction is typically .35, and the number of bytes written is typically 0.2. Using a cost of 4 cycles per byte of instruction fetched and 3 cycles per byte of operand refer-

enced (reasonable for the 8080), this implies that a typical instruction of 1.6 bytes requires 6.4 cycles for the instruction fetch and execution, but only 1.6 cycles for the operand references.

5.1.6 Summary

The 8080 is a worthwhile computer to examine for two reasons: (1) because the primitive instruction set makes an interesting comparison to the 370, and (2) because, by count, there are more 8080 computers being used than any other computer ever built; its numerical superiority justifies our interest.

The first observation is that in many respects it is indistinguishable from larger machines. There seem to be certain invariants -- such as the number of instructions between successful branches -- which hold for at least representative architectures at both extremes. The second observation is that, in many ways, the 8080 is an "incomplete" architecture, and the results of that incompleteness, unlike for the 370, appear clearly in the measurements. There is reassurance, however, in the fact that the 8080 is an experiment in LSI implementation of computers, and will quickly be overtaken by much improved processors as the industry progresses.

5.2 PASCAL P-MACHINE

Both of the machines so far considered have been conventional register-oriented computers, and the similarity in measured characteristics may be due primarily to the similarity in architecture. To choose a radically different machine organization on which to make the same kinds of measurements, we turn to the hypothetical P-machine used in implementation of the Zurich PASCAL P-Compiler [NOR]. The P-Machine is a stack computer with no registers others than those necessary to maintain the stack and the program counter. P-Code is generated by the compiler and can be used directly as a vehicle for intepretive execution, or as an intermediary in the production of machine code for other computers. The measurements made here are for code generated by the SLAC/Stanford version of the compiler [HAZ].

5.2.1 Opcode Distribution

The static opcode frequencies for P-machine code are most reminiscent of those for the 8080 microprocessor (see Figure 19), but the distribution is more sharply skewed. The top two instructions, which account for an incredible 35% of all instructions in the program, are the "load variable" (LOD) and "load constant" (LDC) opcodes. The procedure call opcodes (MST/CUP) are also, as for the 8080, rather high in the static distribution: 7.15%. The

ubiquitous jump is, as usual, among the top few; here the combination of conditional (FJP) and unconditional (UJP) jumps account for 11.02% of all opcodes.

5.2.2 Opcode Pairs

Table 15 shows the first few static opcode pairs in the list ordered by decreasing frequency. Most of the pairs have little architectural significance except for those which are used for conditional jumps (EQU-FJP, for example).

5.2.3 Branch and Execution Distances

The distribution of static jump distances for P-code is even more strongly biased towards small distances than the other machines we have looked at. For the conditional jump (FJP) 62.8% of the jumps are to targets which are 16 or less instructions ahead of the jump; 75.7% are to targets 32 or less away. For the unconditional jump (UJP) the equivalent numbers are 39% and 50%. Almost all (93.7%) of the conditional jumps are forward; they are produced primarily to branch around THEN clauses. Unconditional jumps have backward targets about one third of the time; most of these are loop terminators.

5.2.4 Summary

In some ways, looking at the P-Code generated by the PASCAL compiler has many of the same disadvantages as other

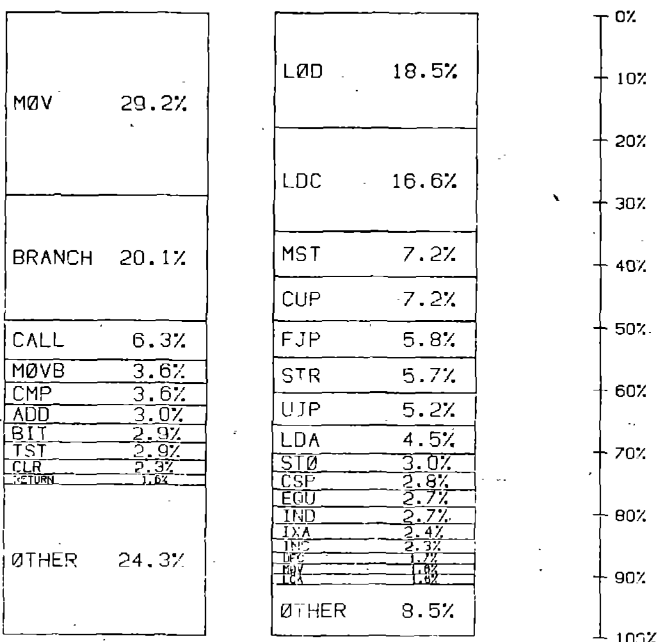


Figure 19: PDP11 and Pcode Static Opcode Distributions

62

Table 15.

Pascal P-CODE Opcode Pairs

First Opcode	Second Opcode	Percent Occurence
LOD	LDC	4.41
MST	LDC	3.00
FJP	MST	2.70
STR	LOD	2.63
LDC	CUP	2.56
CUP	LOD	2.42
LOD	IND	2.31
LOD	INC	2.20
EQU	FJP	2.17
LDC	EQU	2.09

language-oriented studies ([ALE75] for XPL, for example); it tells more about the compiler than it does about the instruction set. P-code is interesting, however, because it describes a stack machine, and the influence of that architecture is seen by the high fraction of load and load constant instructions.

The other lesson to be learned is that it is likely that various special processors or languages will contradict some of the more general findings. One such example is the predominance of forward jumps among the P-Code conditionals; this is a peculiarity of the code generation scheme and it would be dangerous to use such information to design branch mechanisms unless nothing but P-Code programs were to be run.

5.3 DEC PDP11

The PDP11 is an interesting architecture to study because of the regularity and operand symmetry of the instruction set compared to the other machines we have been looking at. There is no question that from an aesthetic point of view such regularity facilitates programming because any reasonable combination of opcode and operand addressing modes is valid. The question remains, however, whether that full generality is exploited in practice.

5.3.1 Opcode Distribution

Figure 19 shows the static distribution of PDP11 opcodes derived from an analysis of about 10,000 lines of code from one of the operatihg systems (RSX-11M). Because of the symmetrical nature of the instruction set, all load store and move instructions are subsumed under the MOV and MOVB opcodes, which accounts for their predominance. The high percentage of branch instructions (20%) is familiar, but the unusually high number of CALLs (6.3%) is due to the highly modular structure of the operating system. In order to determine more about the use of the instruction set it is necessary to examine the way in which the addressing modes are used.

5.3.2 Addressing Modes

For each operand in most PDP11 instructions there is a 3-bit field to indicate the addressing mode, and another 3-bit field which specifies a register to be used. In addition to the 8 addressing modes, special effects for immediate operands can be obtained by using the program counter (PC) as a register; these should be counted as separate addressing modes because they are fundamentally different operations (and are written in assembly language with special syntax.) There are then 12 distinguishable operand addressing modes, and table 16 shows how often they were used in our sample.

Table 16.

PDP11 Addressing Modes - All Operands (Static)

Mode	Syntax	Frequency
Register	R	31.992%
Register indirect	(R)	8.703%
"Push"	(R)+	9.863%
"Push" indirect	@(R)+	.337%
"Pop"	-(R)	4.979%
"Pop" indirect	@-(R)	.026%
Indexed	E(R)	17.312%
Indexed indirect	@E(R)	.553%
Immediate	#E	14.896%
Absolute	@#E	.026%
PC relative	E	11.105%
PC relative indirect	@E	.202%

The four most common modes are perhaps the four simplest: Register, Indexed[1], Immediate, and PC relative (normal "direct addressing"). These four account for 75.3% of all operand references, and are the basic set needed for any efficient instruction set. The frequent use of immediate operands is particularly telling for the 370 architecture, for which the lack of immediate operands causes a space and time penalty to be paid for access to literal data which cannot be placed with the instruction.

The four least-used addressing modes are precisely the four memory-indirect references, accounting for only 0.9% of the operands. Their disuse in an architecture which is so limited by the number of opcode bits (only 15 two-operand instructions are allowed!) raises serious questions about their inclusion. An extra 2 bits of opcode space (1 for each operand) could be used to great advantage, and the penalty of extra instructions when memory-indirect is needed would be small indeed.

It may well be that the stylized code generation of some language processors makes much heavier use of memory-indirect addressing, perhaps for parameter accessing. One must question, however, whether their use is the result of

---

[1] The heavy use of indexed addressing is in part due to the frequent use of record-like data structures within the operating system; indexed addressing is used for field references within the record.

forcing the coding conventions so that efficient instructions could be used, rather than designing a mechanism appropriate from the point of view of the language implementor. If this is the case, even greater efficiency might be gained by removing the otherwise unused addressing modes to allow new instructions, and implementing higher-level primitives which directly match the language requirements. This must be done with careful cooperation between language and machine designers, however; the danger is the design of baroque and beautiful but practically unusable instructions. Wirth [WIR] points to the PDP11 CALL and RET instructions (in other than their simple use) as such an example.

Table 17 shows how the addressing modes are used in the various operand positions, and Table 18 shows the complete two-operand distribution of addressing modes both for all instructions and for the important special case of MOV and MOVB.

The most common single-operand addressing mode by far is the simple address, which the assembler generates as PC relative (unless directed otherwise) so that the code will be relocatable. Full addressing, combined with the other three non-indirect and non-autoincrement modes (Register, Indirect Register, and Indexed), accounts for 86.8% of all single-operand references. The memory indirects account for

Table 17.

PDP11 Addressing Modes - By Operand (Static)

Single-Operand Addressing Modes

Mode	%
R	21.942%
(R)	11.045%
(R)+	7.190%
@(R)+	1.111%
-(R)	3.484%
@-(R)	.074%
E(R)	12.083%
@E(R)	1.037%
#E	.000%
@#E	.000%
E	41.734%
@E	.296%

Two-Operand Addressing Modes

Operand 1		Operand 2	
R	21.709%	R	46.750%
(R)	6.466%	(R)	9.897%
(R)+	12.075%	(R)+	8.841%
@(R)+	.065%	@(R)+	.263%
-(R)	1.517%	-(R)	9.105%
@-(R)	.000%	@-(R)	.032%
E(R)	19.630%	E(R)	17.321%
@E(R)	.362%	@E(R)	.527%
#E	34.147%	#E	2.276%
@#E	.000%	@#E	.065%
E	3.959%	E	4.618%
@E	.065%	@E	.296%

2.2%, and the non-indirect autoincrement modes make up the remaining 10.6%.

It is important to be aware of various idiosyncratic uses of the addressing modes before reaching conclusions about their utility. The relatively high use of autoincrement for one-operand instructions, for example, (7.19%)

65

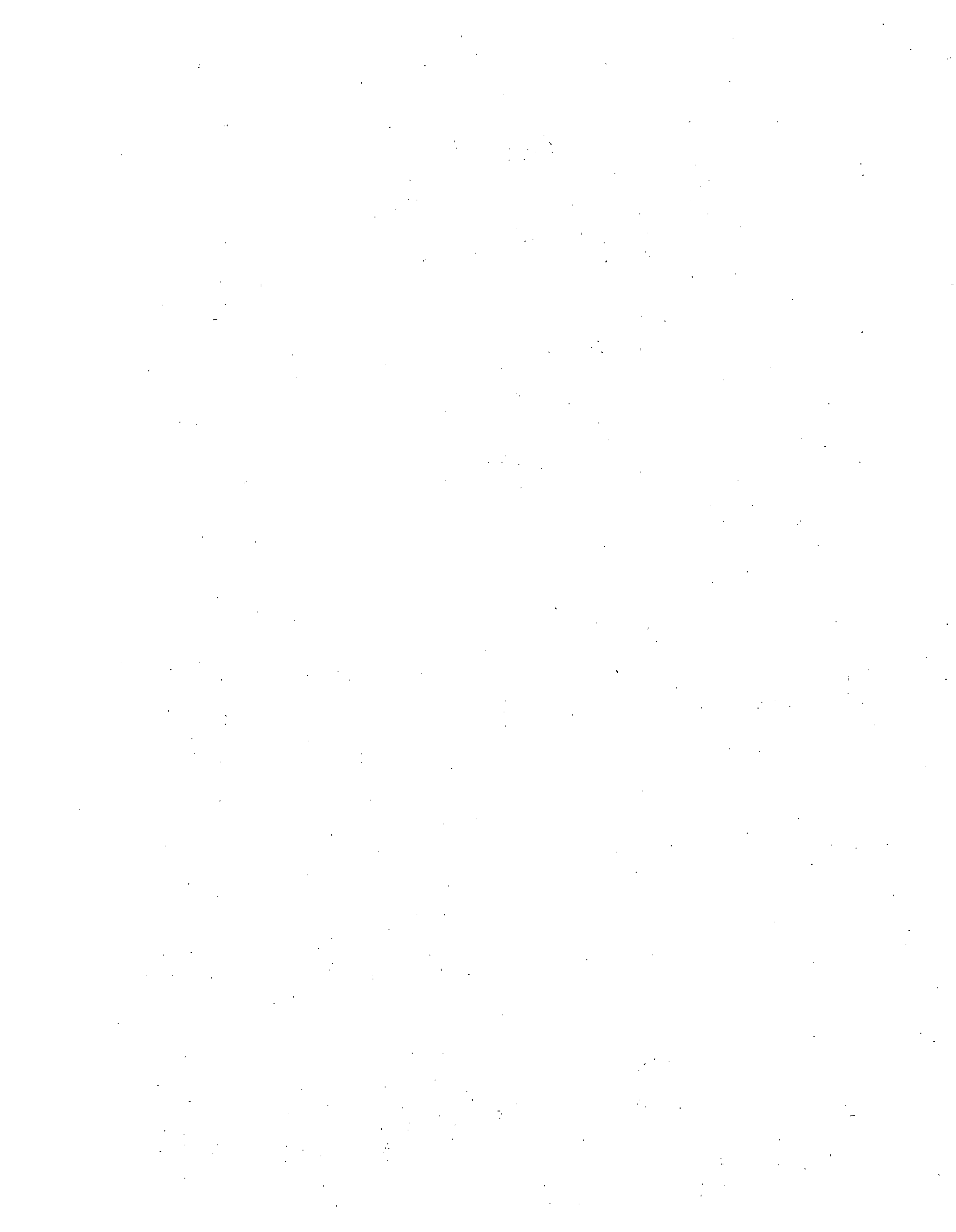


Table 18

PDP11 Two-operand Addressing Modes

PDP11 2-OPERAND ADDRESSING MODE UTILIZATION IN PERCENT (STATIC)

ALL 2-OPERAND INSTRUCTIONS

	R	(R)	(R)+	@(R)+	-(R)	@-(R)	E(R)	@E(R)	#E	@#E	E	@E	SUMS
R	6.697	2.3Ø9	2.21Ø	.Ø33	3.563	.ØØØ	3.827	.165	1.452	.ØØØ	1.287	.165	21.7Ø9
(R)	3.629	.726	.561	.ØØØ	.66Ø	.ØØØ	.528	.ØØØ	.231	.ØØØ	.132	.ØØØ	6.467
(R)+	5.477	1.Ø23	3.497	.165	.429	.ØØØ	.726	.ØØØ	.Ø33	.ØØØ	.693	.Ø33	12.Ø75
@(R)+	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.Ø33	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.Ø33	.ØØØ	.Ø66
-(R)	.561	.132	.Ø66	.ØØØ	.462	.Ø33	.231	.ØØØ	.ØØØ	.ØØØ	.Ø33	.ØØØ	1.518
@-(R)	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ
E(R)	11.316	1.122	1.353	.Ø33	1.815	.ØØØ	2.5Ø7	.231	.495	.Ø33	.726	.ØØØ	19.63Ø
@E(R)	.165	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.198	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.363
#E	16.562	4.322	.792	.ØØØ	1.584	.ØØØ	9.139	.132	.ØØØ	.Ø33	1.485	.Ø99	34.147
@#E	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ
E	2.276	.264	.363	.Ø33	.561	.ØØØ	.165	.ØØØ	.Ø66	.ØØØ	.231	.ØØØ	3.959
@E	.Ø66	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.Ø66
SUMS	46.75Ø	9.898	8.842	.264	9.1Ø6	.Ø33	17.321	.528	2.276	.Ø66	4.619	.297	1ØØ.ØØØ

"MOV" AND "MOVB" ADDRESSING MODES

	R	(R)	(R)+	@(R)+	-(R)	@-(R)	E(R)	@E(R)	#E	@#E	E	@E	SUMS
R	7.689	1.665	2.938	.ØØØ	5.24Ø	.ØØØ	3.722	.245	.ØØØ	.ØØØ	1.665	.245	23.4Ø8
(R)	4.6Ø3	.637	.441	.ØØØ	.833	.ØØØ	.343	.ØØØ	.ØØØ	.ØØØ	.147	.ØØØ	7.ØØ3
(R)+	7.Ø52	1.273	3.673	.147	.588	.ØØØ	.784	.ØØØ	.ØØØ	.ØØØ	1.Ø28	.Ø49	14.594
@(R)+	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.Ø49	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.Ø49
-(R)	.833	.196	.Ø98	.ØØØ	.49Ø	.ØØØ	.147	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	1.763
@-(R)	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ
E(R)	14.887	1.322	1.861	.Ø49	2.595	.ØØØ	3.232	.196	.ØØØ	.Ø49	.784	.ØØØ	24.976
@E(R)	.245	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.294	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.539
#E	13.369	2.644	.441	.ØØØ	2.ØØ8	.ØØØ	2.449	.ØØØ	.ØØØ	.Ø49	1.371	.147	22.478
@#E	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ
E	3.183	.294	.392	.Ø49	.833	.ØØØ	.147	.ØØØ	.ØØØ	.ØØØ	.196	.ØØØ	5.Ø93
@E	.Ø98	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.ØØØ	.Ø98
SUMS	51.959	8.Ø31	9.843	.245	12.635	.ØØØ	11.117	.441	.ØØØ	.Ø98	5.191	.441	1ØØ.ØØØ

is quite misleading. It turns out that 73.1% of those are uses of the TST instruction to increment the register by 2, and are not intended to test the addressed word at all. A common use is "TST (SP)+" to remove a word from the stack. The conclusion is not, therefore, that autoincrement is of significant use for single-operand instructions, but that there should be an instruction which increments by small numbers other than 1.

The two-operand instructions are dominated by the MOV/MOVB instructions, which are used to get the effect of Load, Store, Push, and Pop as well as the more exotic uses. Reading from Table 18, we can reconstruct the following significant special uses of MOV/MOVB:

Table 19.

Special Uses of PDP11 MOV/MOVB

Use	% of MOV/MOVB	% of pgm
Load Register (Including POP, load immediate, and load from reg)	51.9	16.8
Store register (including push)	15.7	5.1
Move simple-address to simple-address	3.2	1.0
PUSH memory word	2.6	.84
Store immediate (indirect register)	2.6	.84
Store immediate (simple address)	2.4	.78
Total	78.4%	25.5%

The other two-operand instructions which are important are CMP/CMPB, ADD, BIT, BIS, BIC, and SUB. Each of them is used primarily in only a few particular ways. Rather than include all the data for these instructions, Table 20 shows the important cases and the percent of that instruction's occurrences which are represented.

Table 20.

Special Uses of PDP11 2-Operand Instructions

Compare Immediate	CMP x,#E or CMP #E,x	51.2%
Compare Register-Indexed	CMP R,E(R)	11.1%
Increment by 4	CMP (R)+,(R)+	10.4%
Add Register-Immediate	ADD R,#E	53.7%
Bit Test Immediate	BIT #E,x	88.8%
Bit Set Immediate	BIS #E,x	61.5%
Bit Clear Immediate	BIC #E,x	66.3%
Subtract Immediate	SUB #E,x	50.0%

5.3.3 Summary

From what we have seen, it certainly appears that the full generality of the PDP11 addressing modes are not used. Many of the instructions are used predominately in "stylized" ways for which less symmetric instruction subsets would be perfectly adequate, and the penalty for substituting more than one instruction for the more exotic cases

is small because their frequency of use is low. This is especially true of the indirect addressing modes, but the case for eliminating any of the other modes is less strong.

What about the symmetry of the two-operand instruction format? Certainly the "ordinary" operations predominate: 68% of the moves are load/store/push/pop of a register, 83% of the compares have one operand that is either a register or an immediate value, and so on. This suggests that perhaps limiting the addressing modes by forcing one of the operands to be either a register or an immediate value would save opcode bits and processor complexity and result in only a modest number of additional instructions.

"Modest", however, is not "insignificant"; for our sample, 9.7% additional instructions would result if only register and immediate two-operand instructions were available[2]. Although the use of each of the remaining exotic combinations of addressing modes is individually quite small, the cumulative effect is non-trivial. The value to be placed on the additional opcode bits depends on constraints imposed by byte or word boundaries and on the need for additional operations. In the case of the PDP11,

---

[2]This assumes that only a single extra instruction is necessary to simulate the missing addressing mode. The 9.7% results from the fact that 19.9% of two-operand instructions would need expansion, and two-operand instructions are 48.7% of all instructions.

the extra 2 bits that would come from removing indirects would seem to make it unnecessary to buy any extra space by destroying the symmetry of two-operand instructions. The final decision, however, is always one of judgment and aesthetics.

One of the other points well illustrated by the PDP11 example is that operations with small integers are very common and warrant special instructions or addressing modes. It is important to be able to increment and decrement address registers efficiently, at least by the size of any of the standard data elements.

Chapter VI
PREDICTIVE MODELS

This chapter will contain a number of examples of models of programs and computer subsections that are either driven by the data generated from interpretive traces, or verified with that data. The examples are presented in increasing order of generality; the first is an example of analyzing the effect of a change to an existing implementation, the second is the analysis of a proposed new implementation, and the third uses some of the measurement results to validate more abstract models of program behavior.

6.1 370 BRANCH IMPROVEMENT

The instruction fetching mechanism is one of the most complex parts of highly pipelined high performance computers. The difficulty is simply that any interruption in the sequential execution of instructions may cause a delay in pipeline sequencing unless the target of the branch is fetched well in advance. Unconditional branches cause a new instruction stream to be fetched, but since the decision to do so can be made as soon as the branch is decoded, no delay will be necessary as long as the memory subsystem can

69

supply the data soon enough. For conditional branches, however, the success of the branch often depends on results of previous instructions which have not been executed at the time that the branch instruction is decoded. Rather than wait for that instruction to complete execution, both the 470 and the 168 make an assumption about the success of the branch, and proceed with instruction decoding accordingly[1].

Of the two machines studied, the 168 uses the more complicated (but still simple) decision mechanism: the success of a conditional branch is predicted on the basis of the opcode and (in the case of BC and BCR branches based on the condition code) the branch mask. Whenever the branch decision can be made without reference to the condition code (BC/BCR with mask 0 or 15, BCTR/BALR with R2=0) the correct decision is made at the time of decoding. All loop branches (BCT, BCTR, BXLE, and BXH) are guessed to be successful, and all other conditional branches (BC/BCR with mask not 0 or 15) are guessed to be unsuccessful. As can be seen from Table 21, the success rate for this procedure is not outstanding.

Some improvement could be obtained by simply changing the choices made. The rarely used BXH probably should be

---

[1]Both machines also prefetch instructions from the alternate path, but those instructions are held in the instruction buffer without being decoded.

Table 21.

Branch Prediction Success For The 168

Program	BCT/BCTR	BXH	BXLE	BC	BCR
FORTC	95.3%	18.1%	97.1%	40.5%	87.0%
FORTGO	87.8%		97.2%	47.2%	43.1%
PL1C	99.7%		99.1%	40.4%	99.9%
PL1GO	90.2%		96.3%	71.5%	27.0%
COBOLC	93.3%		94.9%	40.7%	11.3%
COBOLGO			98.6%	61.4%	55.9%
LINSY2	85.7%		93.4%	44.1%	33.3%
PASCAL	87.5%			54.2%	98.0%
SNOBOL	98.3%		93.6%	26.8%	25.4%
168 Expectation	100%	100%	100%	0%	0%

guessed to be unsuccessful -- which is the choice that seems obvious based on its use as a top-of-loop conditional. The success of the "no branch" guess for the always important BC/BCR is highly program dependent and often close to 50%, so the direction of the guess is almost irrelevant.

In order to improve the branch prediction success, more information about the circumstances of the branch must be involved in the decision. Some of the possibilities are:

1. The direction of the potential branch
2. The distance to the target
3. The condition being tested (the value of the mask for BC/BCR)
4. The preceeding opcode which set the condition code

5. The past history of success for the particular branch instruction

All of these possibilites were examined, but none of them except the last turns out to be a good predictor of the branch success over a large range of programs.

Table 22, for example, shows that the direction of the branch would contribute negligibly to a better guess of success. To assess point 4, all instructions were divided into 18 classes according to basic operation (fixed vs. floating, add vs. subtract. etc.) and the class of the last instruction before the branch which set the condition code was recorded. No program-independent relationships between the condition-setting opcode and the success of the branch were found, except for relatively infrequent and stylized sequences.

Table 22.

Branch Success as a Function of Branch Direction

Program	Forward	Backward
FORTC	37.6%	61.8%
FORTGO	39.5%	83.7%
PL1C	17.2%	90.9%
PL1GO	39.1%	92.0%
COBOLC	36.0%	63.1%
LINSY2	19.8%	94.7%
PASCAL	57.3%	54.7%
SNOBOL	29.0%	12.2%

Although conditional branch instructions as a class are extremely unpredictable, branch instructions individually are remarkably consistent. Figure 20 shows, for PL1GO as an example, that most of the conditional branch instructions (BC or BCR with mask not 0 or 15) either always succeed or always fail; less than 30% of the branch instructions actually go in both directions any time during the history of the program. The second column of Table 23 shows, for all the programs, the fractions of conditional branch instructions which either always succeed or always fail. Not surprisingly, then, the use of the past history of a particular branch is a good estimator of future success. If the guess is that the instruction will branch the same as it did last time, the third column of Table 23 shows that the success of the prediction is consistently above 85% for all programs.

Since the success of the prediction is largely determined by the branches which are heavily executed, it is not the first execution of branch instructions which prevents the prediction success from being even higher. Even some of those heavily executed branches occasionally have prediction successes that are rather low (70%), which contributes strongly toward preventing the overall success from being any higher. Table 24 shows, using the PASCAL compiler as an example, how the average prediction success varies with the number of executions of the branch instructions.

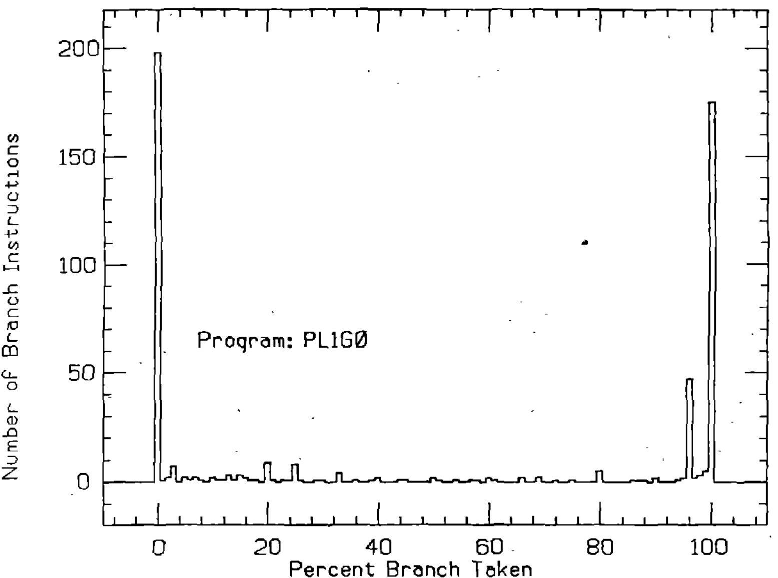


Figure 2Ø: Conditional Branch Successes

The implementation of such a scheme is relatively straightforward; the cache is the appropriate place to record the previous success of the branch, and it can be forwarded to the instruction decoder for use in prediction. If the prediction fails, the new state is written into the cache whenever it is convenient. The results will be somewhat degraded by cache misses which fetch instructions without a history bit, but the effect will be small.

Table 23.

Conditional Branch Consistency and Prediction Success

Program	Consistent Branches	Prediction Success
FORTC	65.Ø%	89.5%
FORTGO	64.1%	86.5%
PL1C	73.Ø%	93.4%
PL1GO	71.4%	92.7%
COBOLC	67.7%	89.7%
COBOLGO	61.9%	96.5%
LINSY2	7Ø.1%	92.9%
PASCAL	7Ø.8%	91.Ø%
SNOBOL	78.7%	97.9%

Table 24.

Branch Prediction Success vs. Number of Executions

Number of Executions	Success
1	54.2%
2-3	76.6%
4-7	77.7%
8-15	86.Ø%
16-31	87.5%
32-63	82.5%
64-127	9Ø.8%
128-255	95.5%
256-511	91.1%
512-1Ø23	71.6%
1Ø24-2Ø47	88.9%
2Ø48-4Ø95	76.9%
4Ø96-8191	99.6%
8192-16383	9Ø.2%
16384-32767	98.7%

Another approach toward implementing a predictive scheme of this type is to mark conditional branch instructions with a prediction before execution; there would essentially be both 'probable' and 'improbable' versions of conditional branches. In most cases the information would

have to be supplied directly or indirectly by the programmer, and although in many cases the choice is clear (the zero-argument test of the SQRT routine, for example) there would be many others that are difficult. Higher-level languages would need a mechanism for getting advice from the programmer -- like the "FREQUENCY" statement in FORTRAN II for indicating the relative frequency of the different exits from a 3-way IF statement. This scheme, while requiring less hardware sophistication, is cumbersome and in almost all cases[2] would result in a lower prediction success than the same-as-before estimate.

Related techniques have been used in other architectures, but usually with limited scope. The Manchester University MU5, for example, [IBB] uses a small associative memory containing the addresses of instructions which caused successful branches, and it is interrogated as each branch is decoded. If the address is found, the associative memory provides the target address of the branch from which instructions will now be fetched instead of continuing sequential execution. Simulation studies indicate that correct prediction occurs in only 75% of the cases, which is significantly less than for the scheme presented here.

---

[2]Exceptions are the rare branches for which the last execution is a poor prediction -- for example a branch which alternately succeeds and fails would do better with a static prediction.

6.2 PREDICTION OF NEW PROCESSOR PERFORMANCE

It is sometimes the case that general-purpose computers are used predominately for special purposes, and it is reasonable to consider whether a specialized processor is not a more cost-effective solution. This section describes one such effort at the Stanford Linear Accelerator Center (SLAC).

There is a computer currently being built at SLAC which will execute a restricted class of 370 programs at speeds up to 1/2 of the 168, yet the cost of the processor is anticipated to be under $10K. Called the "168/E" [KUN], the CPU is made of LSI bit-slice components executing with a 150 ns cycle time, but the traditional microcode overhead for instruction interpretation is avoided by macro-expanding 370 instructions into sequences of micro-orders at compile time. Separate high-speed memories are used for program and data storage to allow two-instruction pipelining. The subset of 370 instructions supported initially include only halfword and fullword arithmetic (sufficient for integer FORTRAN programs compiled by the 370 optimizing compiler) but planned expansion will later include some floating-point instructions.

The only way that such high performance can be obtained from relatively modest hardware is if the match between the load imposed by the software and the efficiently executed

instructions is high. Since the software (physics analysis programs for the LASS Large Aperture Solenoid Spectrometer at SLAC) is already in use on the 168, it was possible to use the trace analysis system described here to predict the performance of proposed designs for the 168/E with almost absolute accuracy.

Figure 21 shows that the distribution of time for the (proposed floating-point version of) the 168/E is quite different from the IBM 168, yet the relatively high fraction of easily-executed instructions results in an overall execution time only 2.1 times that of the 168. This is the result of fine-tuning both hardware and 370-to-168/E translation software to match the characteristics of the application software to be executed.

Note that the attibutes of the LASS software which allowed a special-purpose processor to have such an enormous cost/performance increase over general-purpose processors are not particularly unique. Specifically, the characteristics are:

1. Small, well-defined software available before processor construction
2. Separable program and data space
3. Low use of complex instructions

4. Relatively small data area
5. Modest I/O requirements

These are not attributes that are unique to high-energy physics applications by any means. The 168/E is a good example of how high-performance can be achieved at low cost if the instruction set is kept simple. The difference between a complex instruction set and a simple one is far more significant for the implementation of the processor than for the implementation of the program.

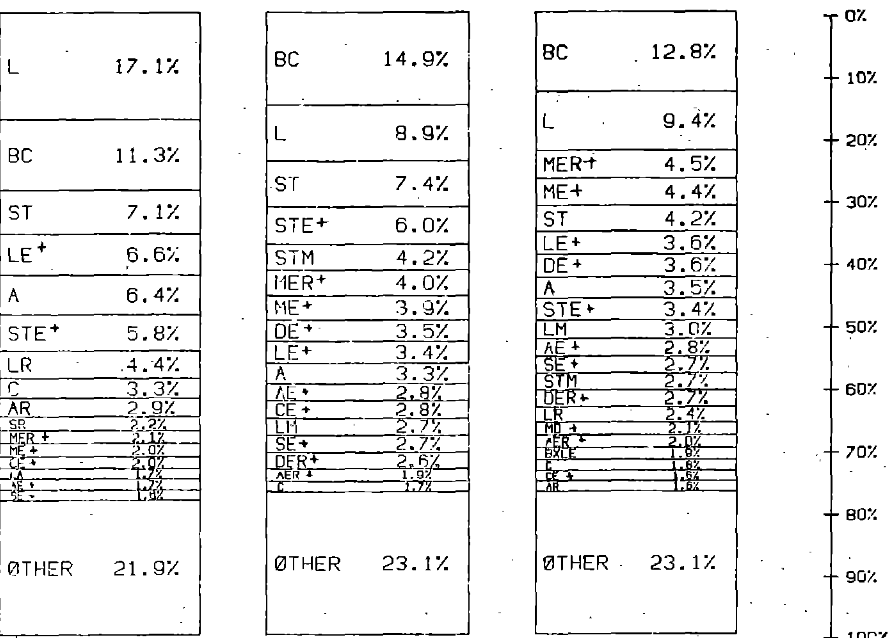


Figure 21: Predicted 168/E Performance Compared to IBM 168

6.3 MARKOV INSTRUCTION-FETCH MODEL

Measurement and analysis of existing or proposed computers is in itself a worthwhile pursuit, but one of important side benefits is the support it gives to abstract models. It is productive to use the results of such measurements to inspire and parametrize simplified models of either programs or computer subsystems. When the models have been analyzed, quantitative comparisons can be made between model predictions and other real measurements so that the model is verified.

The models presented here attempt to abstract and simplify the process of instruction fetching. State-transition diagrams are used to represent simplifications of program behavior, and measured values are used for the transition probabilities. The basic motivation is to derive a model which will predict the execution distance distribution by correctly generating branches imbedded within instruction fetches.

The initial model is based on the measured frequency of branches and instructions of various lengths, and is shown in Figure 22. States S2, S4, and S6 represent execution of 2, 4, and 6-byte instructions with probabilities p2, p4, and p6 respectively, and state B represents the occurence of a branch (with probability Pb). Since for the construction of the execution distance distribution only the occurence of

states S2, S4, and S6 are of interest, the model can be redrawn to eliminate unnecessary states (Figure 23).

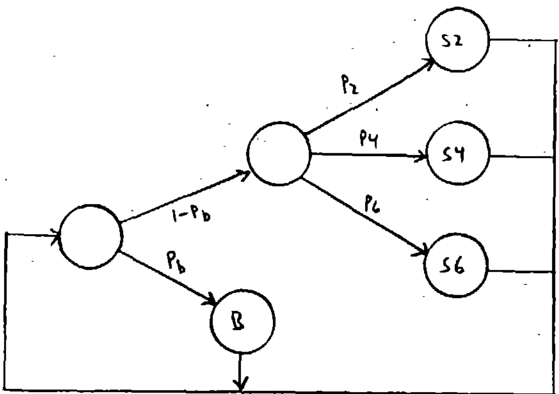


Figure 22: Instruction Fetch Model 1

The execution distance between branches can be computed as a multinomial distribution describing the probability of runs instructions without intervening branches. The probability of executing exactly ni i-byte instructions (for i=2,4,6) is

$$p(n2,n4,n6) = (p2')^{n2} \cdot (p4')^{n4} \cdot (p6')^{n6} \cdot pb \cdot \frac{(n2+n4+n6)!}{n2!\ n4!\ n6!}$$

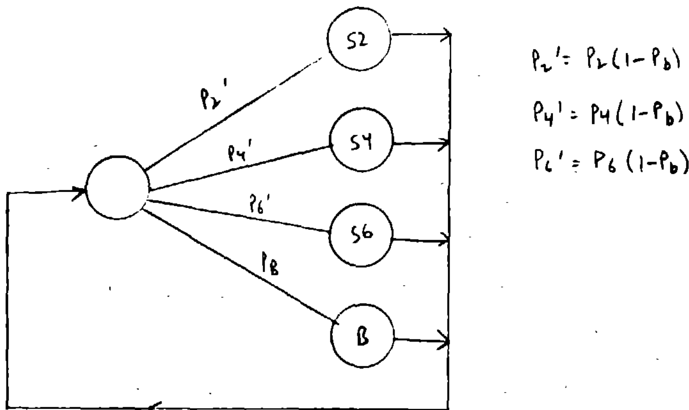


Figure 23: Modified Instruction Fetch Model 1

for which the execution distance is n2+n4+n6. By summing the probabilities of all combinations of n2, n4, and n6 in a bin for the sum n2+n4+n6, the execution distance distribution predicted by the model can be computed. If PD(n) is the probability of an execution distance of n bytes, then

$$PD(n) = \sum_{2n2+4n4+6n6=n} p(n2,n4,n6)$$

which distribution is shown in Figure 24 as Model 1.

In comparison to the actual execution distance distribution of the program from which the probabilities were taken (Figure 24) the model fares poorly,

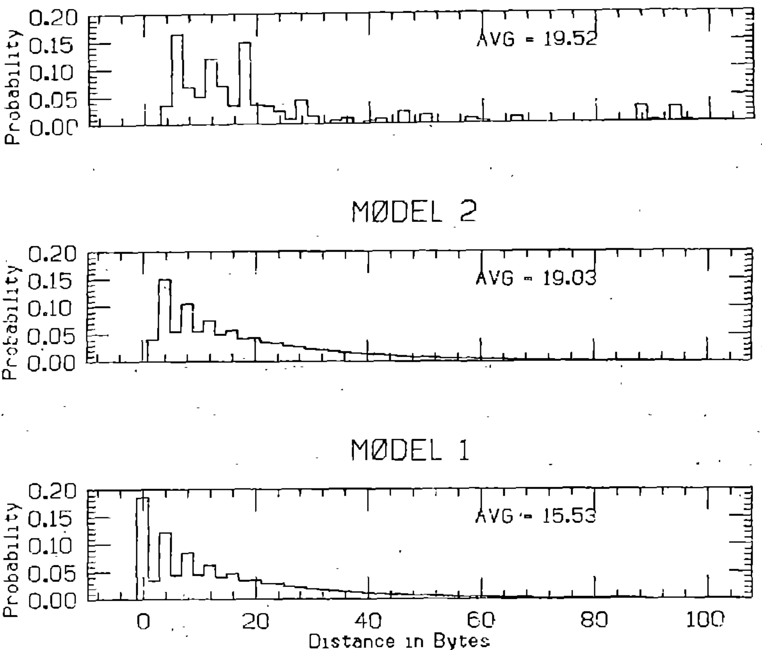


Figure 24: Execution Distance Distributions

mostly because zero-length execution distances (branches to branches) are allowed in the model although they rarely occur in reality. The model of Figure 25 corrects this by forcing a branch to be followed by a true instruction and not another branch. The new execution distance can be computed from the old one by "shifting" it one instruction. If PD' is the new execution distance distribution, then

$$PD'(n+i) = PD(n) \cdot pi \qquad \text{for } i=2,4,6$$

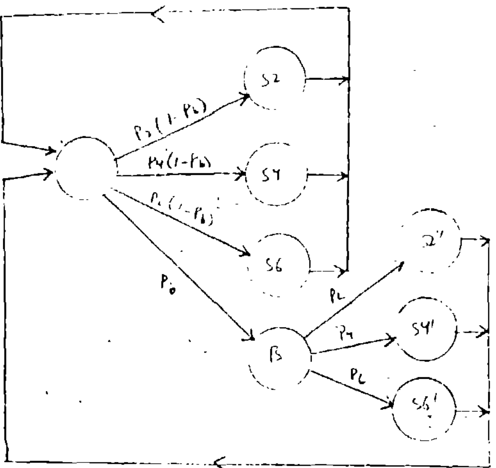


Figure 25: Instruction Fetch Model 2

The result is shown in Figure 24 as Model 2. The match is now better, and the average distance (19.03 bytes) is in better agreement with the true average (19.52 bytes). The details of the true execution distance curve obviously cannot be reproduced by a model with as few parameters as this one.

The distributions and averages so far computed from the models have required sums over combinations of integers; the sums were truncated when the contribution of the terms became negligible. In fact, however, an analytic solution to the model as a Markov chain is possible, and leads to direct computation of the average and variance of interesting quantities, especially the execution distance.

The quantity we can compute directly from the Markov model is the passage time between two states, i.e. the number of intermediate states visited between the two states. If the occurence of a branch is represented by a single state b, and if all other states represent the execution of one byte of non-branch instructions, then the mean passage time from state b back to state b is the average execution distance as we have measured it. This can be accomplished by breaking each state which represents a single instruction into a number of states proportional to the length of the instruction (Figure 26). Note that branch-to-branch transitions have been eliminated, as well as 6-byte branch instructions. The state transition matrix P for this model is

$$\begin{matrix} p2(1-pb) & p4(1-pb) & p6(1-pb) & pb & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ p2 & p4 & p6 & 0 & 0 & 0 & 0 \\ p2(1-pb) & p4(1-pb) & p6(1-pb) & pb & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ p2 & p4 & p6 & 0 & 0 & 0 & 0 \end{matrix}$$

The first quantity of interest is the state probability vector A which gives the steady-state probability of being in each of the states. For any vector a of state probabilities the new vector of state probabilities after one step is aP. The steady-state state probability vector is therefore that vector a which satisfies aP=a. This vector can be shown [KEM] to be unique for any regular

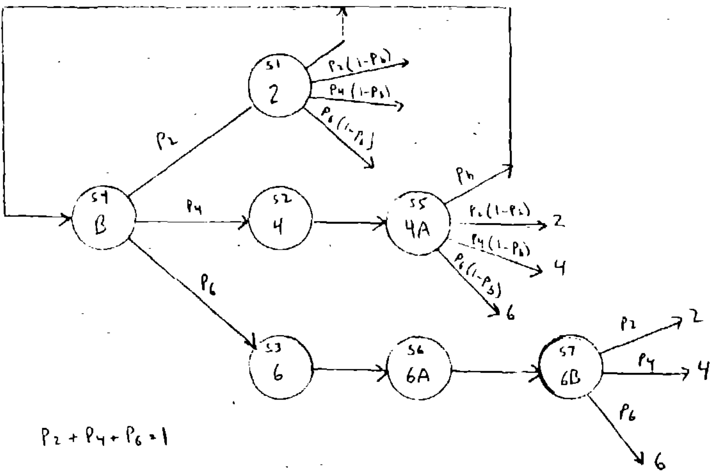


Figure 26: Modified Instruction Fetch Model 2

Markov chain (a regular Markov chain is simply one in which every state is eventually accessible from every other state).

The computation of the mean and variance of the passage time is best done by first computing the "fundamental matrix" of the Markov chain as

$$Z = (I-(P-A))^{-1}$$

where A is the matrix all of whose rows are the state probability vector a and I is the identity matrix. The matrix of mean passage times is then

78

$$M = (I - Z + EZ_{dg})D$$

where Z_{dg} is the fundamental matrix Z with all off diagonal entries set to 0, and D is the diagonal matrix with elements d(i,i)=1/a(i). (For the proof of these expressions see [KEM]). The matrix of variances of the passage time, V, can be computed by

$$W = M(2Z_{dg}D - I) + 2(ZM - E(ZM)_{dg})$$

$$V = W - M^2$$

where M^2 is formed by squaring each element of M.

Table 25 shows the result of these computations for the benchmark jobs compared to the measured values; the agreement is reasonably good in all cases.

Table 25.

Measured vs Predicted Execution Distances

Program	Measured Average	Measured Std. Dev.	Predicted Average	Predicted Std. Dev.
COBOLC	19.86	17.25	21.58	17.91
FORTGO	27.33	30.38	29.54	25.84
PL1GO	69.40	34.11	73.53	69.75
LINSY2	30.42	19.40	30.55	27.11
FORTC	26.04	25.07	26.35	22.57
PL1C	15.94	13.51	17.20	13.52

A model such as this one can be useful as a subpart of larger models of computer systems; it is essentially one of the components of a model which describes program behavior. It is simple enough to be mathematically tractable, but accurate enough to make the use of detailed traces unnecessary. It could, for example, be used as an address generator for modeling instruction pipelines or caches when combined with a similar model for branch distances based on the measured branch distance distribution. The result is a program model which is specified by a small number of parameters (the transition probabilities) derived from measurements.

Chapter VII

ARCHITECTURAL CONCLUSIONS

A large amount of data has been collected in the course of this study, only a part of which has been included and interpreted here. This section will summarize some conclusions, assertions, and allegations that can be derived from the data. Some are obvious but often overlooked, some are surprising, and some -- particularly some of the recommendations that seem warranted -- may be controversial.

Many of the conclusions will be concerned with the effect of architectural features on the implementation of a high-performance processor. The emphasis on performance rather than aesthetics is deliberate. Without an interest in performance the study of architecture is a sterile exercise, since all computable problems can be solved using trivial architectures, given enough time. The challenge is to design computers that make the best use of available technology; in doing so we may be assured that every increase in processing speed can be used to advantage in current problems or will make previously impractical problems tractable.

Just as increasing basic device speed does not reduce the importance of efficient architectures, neither does the trend toward small distributed computers. It is clear that compared to current large computers, the new generation of small computers will be small in physical size only. The CPUs, even if implemented as single-chip LSI microprocessors, will be straining the limits of speed imposed by the technology much more than the limits of complexity, and the same techniques for producing today's high-performance large computers will be required.

7.1 TIME IS OF THE ESSENCE

The basic point here is that instruction frequency distributions, much discussed in the literature, are often dangerously misleading. This is particularly true for the increasingly common architectures which have extremely complex and time consuming instructions as well as the normal complement of simple and fast operations. There is no substitute for careful analysis of both program and processor to determine how time will be distributed.

Many dramatic examples have been shown of instructions which are negligible by static or dynamic count but are very important or even dominant in the processor performance. Because of the emphasis on efficient execution of the simple

instructions, the Amdahl 470 is particularly prone to suffer from this effect. For LINSY, floating point multiplication rises from 3% by count to consume 18% of the time. For FORTC, multiple register loads and stores rise from less than 2% by count to almost 15% by time (for the IBM 370/168). By far the most astounding example is a COBOL program for which a negligible number of executions of decimal divisions consumed 19% (for IBM) or 33% (for Amdahl) of the total program execution time.

7.2 ELIMINATE DECIMAL INSTRUCTIONS

The inclusion of decimal arithmetic is most often justified by the assumption that the cost of conversion to and from binary format is too costly compared to the relatively meagre amount of arithmetic which will be performed. We have seen that no small amount of decimal arithmetic can safely be called meagre, and it would probably be better to eliminate directly computing decimal arithmetic and do the conversions when necessary.

Several points can be made to further support this recommendation.

1. Practically none of the uses of decimal arithmetic we have seen are required because the precision is greater than that afforded by single-precision binary arithmetic; almost all the decimal

operands are 4 bytes (7 decimal digits) or less. Even if this were not the case, the argument would then be to include (say) double-precision binary arithmetic -- not long precision decimal arithmetic.

2. There is a space penalty to be paid for the decimal encoding, as well as the time penalty. At the byte level the penalty is about 20% (4 bits instead of 3.32 bits needed for digits 0 to 10) but due to the treatment of byte boundaries and the sign, the practical penalty can be 50% (it takes 6 bytes to express a 32-bit binary number). There have been some proposals for a more efficient decimal encoding [CHE], but binary encoding is still optimal as well as computationally simple.

3. As the speed and capacity of computer systems increases, a smaller and smaller fraction of the computation is performed on data which will be directly viewed. Much of the manipulation, particularly of the type for which decimal arithmetic is currently used, is for database information which is being searched, summarized, or organized, and only a small part of the data will be (or indeed could be!) produced for examination

by humans at any one time. Avoiding the use of decimal encoding within the database results in a significant savings in space as well as an important reduction in conversion to and from binary which would have been necessary for processors without decimal arithmetic.

7.3 STRING MOVES ARE DANGEROUS

There is a class of instructions, of which the MVC byte move instruction is a good example, whose members suffer from from the "overhead dominance" effect. These instructions typically process variable length operands at an efficient rate per element compared to the equivalent program loops, but have a relatively expensive startup overhead. This is not necessarily bad, but the effects become unpleasant if the typical use is for short operands, in which case the execution time is dominated by the overhead. Such is the case for MVC; it is typical for the median string length to be near 4 bytes and is often word-aligned. Figure 27 shows that MVC is almost the worst possible way to move such short data.

The solution is to avoid MVC for moving short operands whose length is known at compile time (the majority of the cases) and to have the compiler -- or assembler macro-generator -- produce the code which is efficent for the operand. The remaining cases of MVC are either for long

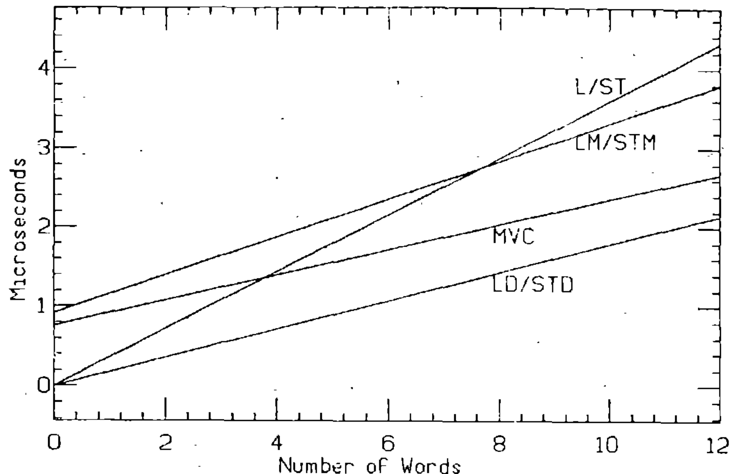


Figure 27: Alternate MOVE schemes

operands (which are not overlapped: see the next section) or for variable length operands. Those cases, however, represent such a small fraction of the time that they may be safely implemented as a call to a short subroutine or an inline expansion of a small loop. The conclusion, therefore, is that MVC should be eliminated.

This recommendation is one of a series whose common theme is that the determination of special cases should be done in software rather than hardware whenever possible. This simple form of very local optimization, when done once by the compiler, frees the processor from making the same optimization repeatedly as it tries to keep several million instructions executing per second. In some cases the

decisions made by the compiler will depend on the particular computer model to be used, but it doesn't seem at all inconsistent for the low-level code generators to use model-dependent information to advantage when it is available. In most cases, however, the decisions are not strongly model-dependent, and optimizations designed for the high-speed models will not generally have a negative effect for the other models.

A frequently-voiced objection to this proposal is that it is somehow "aesthetically unpleasing" to design a machine without a uniform mechanism for simple operations like string movement. This lack, however, is certainly invisible from any moderately high-level programming language, and can be made so for assembly-language programmers by judicious use of macros (which are already becoming popular for introducing reasonable control structures to low-level programming). The day of mandatory one-to-one correspondence between "primitive" operations and machine instructions is over. The fastest single-processor general purpose computer currently available does not even have a "divide" instruction as a single primitive [CRA].

7.4 WATCH OUT FOR UNUSUAL SPECIAL CASES

It is unavoidable that any sufficiently rich architecture will have instructions for which special cases can be used for unusual purposes. It is sometimes the intention

of the designer that they be so used, but it is sometimes an accident of regular design that was not specifically included. It is important to search for these cases (and to study them retrospectively) to determine whether it would be prudent to either (1) provide a new architectural feature to substitute for a perverse special case, or (2) take special care for efficient implementation.

A good example where the first course of action would have been recommended is the use of the two-operand auto-increment TST instruction on the PDP11, where 73% of the cases examined were TST (SP)+ used for incrementing the stack pointer by 2, making unnecessary memory references. It is clearly desirable to have instructions which increment and decrement registers by small values, especially for registers used primarily to hold addresses.

Another example, but one for which an efficient implementation can be achieved once the special case is recognized, is the 370 Exclusive-Or Character ("XC") instruction. Almost all cases ever encountered in any of the programs traced had identical first and second operands; the instruction is used to simulate the missing "store zero" instruction. The IBM 370/168 recognizes this case and efficiently executes a doubleword at a time, so that XC rarely appears as a significantly time-consuming instruction. For the Amdahl V6, however, the "store zero"

XC is identified as one of the awkward cases of overlap between the first and second operands, for which processing must proceed a byte at a time to insure correct execution. The XC instruction implemented in this fashion takes almost 10% of the execution time of one of the traced programs ("COBOLC"). Another example of this type is the overlapped move character instruction used for blanking or zeroing fields.

7.5 INDIRECT ADDRESSING - NO!

In the only architecture studied which has memory indirection as a basic addressing mode (PDP11) it seemed clear that the frequency of use did not justify its inclusion. Other data for the PDP10 [AGA73] confirm this conclusion.

The inclusion of a particular addressing mode -- especially one which, like indirect addressing, can be so easily constructed with an extra instruction -- must have strong quantitative justification. The penalty paid in wasted bits for an used mode is high, since it appears with every instruction. The two bits wasted in the PDP11 for indirect addressing modes, for example, could have been better used to provide either additional two-operand instructions or an additional eight registers.

7.6 IMMEDIATE ADDRESSING - YES!

Immediate addressing was heavily used in all the architectures which include it; for example 34% of all first operands in PDP11 two-operand instructions are immediate. There can be no question that it is one of the more successful architectural features.

It is, incidentally, a feature for which the harmful effects of its absence are much harder to measure than the benficial effects of its inclusion. The 370 architecture has no immediate operands larger than 1 byte, but it is not possible to distinguish which uses of other addressing modes would actually have been immediate. The penalty, both in space and time (for addressing and accessing an out-of-sequence literal) is certain to be significant.

7.7 BEWARE OF EXPENSIVE SOFTWARE CONVENTIONS

The time-consuming effect of some unnecessary software conventions are easy to detect with even simple measurements. The most outrageous example encountered was the program PL1GO, for which 16% of the instructions executed were Move Byte Immediate (MVI), representing 23% of the execution time on the 168, and 13% on the 470. What makes this outrageous is that essentially all occurences are used to record the statement number of the source program which produced the code about to be executed, so that an intelligible traceback can be printed if a fault occurs. (In fact,

two MVI instructions are used if the statement numbers exceed 255!)

There are obviously more efficient ways to get the same information (a table of statement offsets, for example) and no doubt an alternative would have been used if the language implementors knew what the expense of their solution was. Other examples (although less dramatic in effect) are the use of NOP instructions in FORTRAN-generated code to record source statement numbers, and the operating system convention which requires a branch around the character name at the entry to every routine.

The moral is simple: The effect of software conventions should be carefully measured, especially when alternatives exist.

7.8 ARCHITECTURAL INCONVENIENCES CAN BE OVERCOME BY COMPILER OPTIMIZATION

There are some features (or absence of features) in particular architectures which are initially perceived to be awkward, but can often be overcome by clever compilation techniques. An example, for the 370, is the lack of indexing modes or instructions which implicitly multiply the index by the length of the operand. The naive compiler is therefore forced to explicitly multiply (often by shifting) in order to compute the byte address of a subscripted variable. This not only takes an extra instruction, but destroys the register copy of the index.

In many cases, however, identification of the loop structure reveals that the index variable is used only as a subscript and can be replaced by a scaled version of the variable within the loop. The extra shift or multiply instructions may then be removed. The IBM Fortran-H compiler does this optimization, and the effect can be seen from Figure 9 which shows the dynamic instruction counts for a program compiled under increasing levels of optimization. At the "OPT=1" level the shifts to multiply by eight for double-precision floating point arrays ("SLL") are still 8.6% of the instructions, but at the "OPT=2" they have all but disappeared.

This is not to say that more effective indexing modes or instructions are not important. There are other cases, particularly where the index is used both as a subscript and in a non-subscript expression, where this optimization may not be possible. The point here is that moderately sophisticated compiling techniques can overcome inadequacies in the architecture for most of the simple cases.

7.9 THE IMPORTANCE OF BRANCHES

It is clear from almost any study of instruction frequencies that branch instructions are crucial, and the point won't be belabored here. Suffice to say that any implementation with inefficient execution of either successful or unsuccessful branches will suffer substantially for it.

85

It is significant to note that there are few program-independent measures that can characterize branches in more detail. In particular, the fraction of conditional branch success and the branch direction vary considerably and cannot be used to optimize branch implementation. Although certain idiosyncratic code generation may result in more predictable branches -- like the predominance of forward conditionals in the PASCAL P-Code -- it would be dangerous to design a machine with that assumption unless only P-Code is to be run. A design prepared for a wide variety of branching patterns is necessary for general purpose machines, but much can be gained from adaptive schemes such as the branch history mechanism described in section 6.1.

7.10 THE CONSEQUENCES OF SMALL EXECUTION DISTANCES

The high frequency of successful branch instructions results in remarkably short execution distances. For all the architectures examined the average execution distance betweeen successful branches was typically between 5 and 12 instructions; this is one of the few universal characteristics of all programs for all machines.

The more primitive instruction sets did not produce longer instruction sequences in order to compose complex operations out of simpler ones. Either that composition requires that branch instructions be embedded within the sequence, or the units of work being performed are themselves simpler.

The execution distance has a number of important consequences for instruction prefetch and cache design, both of which contribute to having instructions available before they are needed for decoding and execution. The amount of prefetch and especially the cache line size should be commensurate with the number of bytes of instructions that will be executed before a branch causes the prefetched data or the data in the cache line to be abandoned. Making the cache line size much smaller than that amount will cause both pipeline breaks and excessive memory traffic to fetch subsequent instructions. Making the line size too large wastes valuable space in the cache with instructions which are not executed. The choice of 32 bytes (8 average instructions) for the cache line size of both the 168 and the 470 is justified on this basis.

Since the cache line must be aligned, some inefficiency will occur when the target of a branch is near the end of the line. This occurs for two reasons: first, unneeded data (from the start of the line to the target of the branch) will be fetched and will displace other cache data. Secondly, the prefetch effect will be small, since another line access and perhaps a cache miss will likely occur before another branch is taken. Rather than increase the line size to reduce this occurence, a better scheme is to prefetch a second line when a branch occurs to an instruction near the end of a line but not when the branch

is to an instruction near the beginning of a line. The 470 has an optional mode wherein cache lines are prefetched, but it is triggered by any instructions executed near the end of a line, not just those for the beginning of an execution sequence. It is not successful because the unnecessary fetches which occur when an execution sequence terminates near the end of a line outweigh the benefit from the prefetch of sequences which start near the end of the line.

7.11 ADDRESS DISPLACEMENT SIZE SHOULD BE LARGE

Unlike immediate constants, displacements in address expressions are not predominately small values. Although encoding efficiency can be gained by allowing variable-length displacement fields, it is a mistake to limit the displacement value to anything less than the full size of an address.

7.12 ARE THERE MISSING INSTRUCTIONS?

One technique for detecting "missing" instructions is to examine sequences of executed instructions to see if any combinations occur frequently enough to warrant their combination as single instructions. Such sequences do not appear in the "mature" instruction sets like the 370. The sequences which are frequent are of two kinds:

1. Logical pairs which would save only a single opcode pair by being combined, since all other

fields would be necessary. Test-branch pairs are an example.

2. Sequences which are the result of the particular inner loops dominating a program. These have no particular architectural significance.

For the "immature" microprocessor architectures, however, there are several examples of instruction sequences which could profitably be combined (see section 5.1.2). They have not yet reached the level at which single instructions are well-matched to the fundamental operations to be performed.

7.13 THE EFFECT OF TASK SWITCHES ON CACHE CONTENTS

There are two complementary facts about cache memories which are quite clear: (1) a modest cache (16K bytes for example) is sufficient to provide a hit ratio well over 90% for almost all programs, but (2) the effectiveness of the cache is easily destroyed by frequent task switches, interrupts, and supervisor calls. The measurements made here show that, for supervisor calls at least, the entire contents of the cache are often destroyed each time, and the same is likely to be true for interrupts and task switches.

One solution to this dilemma is to establish multiple caches and manipulate the ownership of the cache as part of

the state information of a process. Shared data would have to be prevented from becoming resident in a "private" cache, so two caches might be active at any one time - the system cache containing shared data and the currently active "user" cache of private data. (The page table entries could contain a record of ownership for use in deciding which cache to use.) Note that since only two caches are active at a time, the hardware complexity for an N-cache system is not N times that for a single cache system, since the accessing mechanism (the associative search and replacement logic) can be shared among all the user caches. In a way the storage part of the caches share the accessing logic in the same fashion that peripheral processors share the control logic in the CDC 6600, except that rotation of the "barrel" occurs between task switches rather than between instructions. The hardware that must be duplicated for each user cache is only the data part, which is highly integrated and therefore both small and cheap in the quantities required (16K to 64K bytes, say).

This is only one of many possible schemes for dealing with the problem of degraded cache efficiency in multiprogrammed systems, and certainly would need further study to justify. The existence of the problem is clear, however.

7.14 EPILOGUE

It would be wonderful if, as a result of the measurements and analysis done here, a manual could be produced which would describe the procedure to be followed to design a complete and efficient instruction set. Unfortunately it is impossible to do so; designing a computer is as much a mixture of science and art as is architecting a building or, indeed, writing a computer program. Cleverness and intelligent intuition is a necessity, but both of those facilities can be developed and improved as a result of careful study of the failures and successes of previous designs.

REFERENCES

[AGA73] Agarwal, D.P., "Design of an Efficient Instruction Set", Carnegie-Mellon, 12/72, 11/73

[AGA75] Agajanian, A.H., "A Bibliography on System Performance Evaluation", Computer, November 1975, pp. 63-74.

[ALE72] Alexander, W.G., "How a Programming Language is Used", Computer Systems Research Group, University of Toronto, Report CSRG-10, February 1972.

[ALE75] Alexander, W.G., Wortman, D.B., "Static and Dynamic Characteristics of XPL Programs", Computer, November 1975, Vol 8, 11, pp. 41-46.

[AMD] Amdahl 470V/6 Machine Reference Manual, Amdahl Corporation, Form No. MrM 1000-1, 2nd Ed., 1976, Sunnyvale, Calif.

[ANA] Anagnostopoulos, P.C., Michel, M.J., Sockut, G.H., Stabler, G.M., VanDam; "Computer Architecture and Instruction Set Design", NCC 1973, pp. 519-527.

[ARB] Arbuckle, R.A., "Computer Analysis and Throughput Evaluation", Computers and Automation, January 1966, pp. 12-15.

[BAT] Batson, A.P., Brandage, R.E., and Kearns, J.P., "Design Data for Algol-60 Machines", Proc. 3rd Symposium on Computer Architecture, IEEE, 1976, pp. 151-154.

[BEN] Bencher, D., "OS/VS2 Release 1 Functional Description", SHARE XL Proceedings, March 1973, pp. 320-324.

[BOW] Bowra, J.W., and Torng, H.C., "The Modeling and Design of Multiple Function-Unit Processors", IEEE Transactions on Computers, Vol C-25, No 3, March 1976, pp. 210-221.

[BRO] Bron, "A PASCAL Compiler for the PDP11", Software - Practice and Experience, Vol 6, 1976, pp. 109-116.

[CHE] Chen, T.C., and Ho, I.T., "Storage-efficient Representation of Decimal Data", Communications of the ACM, Vol 18, No 1, January 1975.

[CON] Connors, W.D., Mercer, V.S., Sorlini, T.A., "S/360 Instruction Usage Distribution", IBM Systems Development Division, Report TR 00.2025, Poughkeepsie, N.Y., May 1970.

[CRA] CRAY-1 Computer System Reference Manual #2240004, Cray Research, Inc., 1976

[CUR] Curnow, H.J., and Wichmann, B.A., "A Synthetic Benchmark", Computer Journal, Vol 19, No. 1, February 1976, pp. 43-50.

[ELS76a] Elshoff, James L., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, Vol SE-2, No 2, June 1976, pp. 113-120.

[ELS76b] Elshoff, James L., "A Numerical Profile of Commercial PL/I Programs", Software - Practice and Experience, Vol 6, 1976, pp. 505-525.

[EME] Emery, A.R., Alexander, M.T., "A Performance Evaluation of the Amdahl 470V/6 and the IBM 370/168", CMG IV, October 1975, San Francisco.

[FLY] Flynn, M.J., "Trends and Problems in Computer Organizations", Information Processing 74, North Holland Pub. Co., pp. 3-10, 1974.

[FOS71a] Foster, C.C., Gonter, R., "Conditional Interpretation of Operation Codes", IEEE Trans. on Computers, January 1971, pp. 108-111.

[FOS71b] Foster, C.C., Gonter, R.H., Riseman, E.M., "Measures of Opcode Utilization", IEEE Transactions on Computers, May 1971, pp. 582-584.

[FUL] Fuller, S.L., and Burr, W.E., "Measurement and Evaluation of Alternate Computer Architectures", IEEE Computer, Vol 10, No 10, October 1977, pp. 24-35.

[GEN] Gentleman, W.M., and Wichmann, B.A., "Timing on Computers", Computer Architecture News, SIGARCH, Vol 2, Oct 1973.

[GIB] Gibson, J.C., "The Gibson Mix", IBM System Development Division, Report TR 00.2043, Poughkeepsie, N.Y., 1970. Research done in 1959.

[HAN] Haney, F.M., "Using a Computer to Design Computer Instruction Sets", Carnegie-Mellon, May 1968 PhD Thesis.

[HAZ] Hazeghi, S. , "The SLAC 370 PASCAL Compiler", Computation Group Technical Memo (in progress), Stanford Linear Accelerator Center.

[HEH74] Hehner, E.C.R., "Matching Program and Data Representations to a Computing Environment", Computer Systems Research Group, University of Toronto, Report CSRG-44, November 1974.

[HEH76] Hehner, E.C.R, "Computer Design to Minimize Memory Requirements", IEEE Computer, August 1976, pp. 65-70.

[HEH77] Hehner, E.C.R., "Information Content of Programs and Operation Encoding", Journal of the ACM, Vol 24, No 2, April 1977, pp. 290-297.

[HUG] Hughes, J.H., "A Functional Instruction Mix and Some Related Topics", International Symposium on Computer Performance Modeling Measurement and Evaluation, Cambridge, Mass., March 1976.

[IBB] Ibbett, R.N., "The MU5 Instruction Pipeline", The Computer Journal, 15, 1, pp. 42-50.

[IBM70] IBM System/370 Principles of Operation, Form No. GA22-7000, IBM Corporation, Poughkeepsie, N.Y., 1970.

[IBM74] IBM System/370 Model 168 Theory of Operation / Diagrams Manual, Form No. SY22-6931-6936, Volumes 1-6, IBM Corporation, Poughkeepsie, N.Y., 1974.

[INT] Intel 8080 Microcomputer Systems User's Manual, Intel Corp., Santa Clara, 1975.

[JAY] Jay, R.M., National CSS Inc, Distribution at SHARE, New York, August 1975.

[KAP] Kaplan, K.R, Winder, R.O, "Cache-Based Computer Systems", Computer, March 1973, pp. 30-36.

[KEM] Kemeny, J. G., and Snell, J. L., "Finite Markov Chains", Chapter 4, Van Nostrand Reinhold Company, New York, 1960.

[KIL] Kildall, G.A., "High-level Language Simplifies Microcomputer Programming", Electronics, June 27, 1974, pp. 103-109.

[KNU] Knuth, Donald E., "An Empirical Study of FORTRAN Programs", Software - Practice and Experience, Vol 1, 1971, pp. 105-133.

[KUM] Kumar, B., "Performance Evaluation of a Highly Concurrent Computer by Deterministic Simulation", University of Illinois Coordinated Science Laboratory Report R-717, February 1976.

[KUN] Kunz, P., "The LASS Hardware Processor", Nucl. Instrum. Methods, Vol. 135, pp. 435-440, 1976.

[LIP] Lipp., H., "Instruction Timing for the CDC 7600 Computer", European Organization for Nuclear Research, CERN 75-19, Geneva, December 1975.

[LEW] Lewis, J.B., "Benchmarking and Hardware Monitoring the Amdahl 470 Computer in a Batch Environment", Share CME Newsletter No 42, March 1977, pp. 4-8.

[LUN74] Lunde, A., "Evaluation of Instruction Set Processor Architecture by Program Tracing", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., July 1974.

[LUN77] Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures", CACM, 20, 3, March 1977, pp. 143-152.

[LYO] Lyon, G.E., "Static Language Analysis", National Bureau of Standards Technical Note 797, October 1973.

[MAR] Martin, O., "Performance Measurements on the 7600 at CERN", CERN Technical Report DD/76/17, October 1976.

[MER] Merrill, B., "370/168 Cache Memory Performance", SHARE Computer Measurement and Evaluation Newsletter, July 1974, pp. 98-101.

[MUR] Murphey, J.D., and Wade, R.M., "The IBM 360/195 in a world of Mixed Job Streams", Datamation, April 1970, pp. 72-79.

[NEW] Newman, M.J., "Evaluation of a Processor's Performance by Modeling", M.S. Thesis, Dept. of Elec. Eng. and Comp. Science, Massachusetts Institute of Technology, August 1976.

[NOR] Nori, K.V., et al, "The PASCAL <P> Compiler: Implementation Notes", Eidgenossische Technische Hochchule, Zurich, 1974.

[PEU77a] Peuto, B., Shustek, L. "An Instruction-set Timing Model of CPU Performance", Proc. Fourth Ann. Symp. on Computer Architecture, March 23-25 1977, IEEE.

[PEU77b] Peuto, B., Shustek, L., "Current Issues in the Architecture of Microprocessors", IEEE Computer, V10, No. 2, February 1977, pp. 20-25.

[RAF] Rafii, Abbas, "Empirical and Analytic Studies of Program Reference Behavior", Ph.D. Thesis, Stanford University, 1976.

[RAN] Randell, B., and Russel, L.J., "ALGOL 60 Implementation", Academic Press, London, 1964.

[ROE] Roehr, K., and Niebel, K., "Proposal for Instruction Time Objectives", Performance Evaluation Review, Vol 5, No 2, April 1976, pp. 11-18.

[ROS] Rossmann, G.E., Palyn Associates, unpublished communication.

[SAA72] Saal, H.J., and Shustek, L.J., "Microprogrammed Implementation of Computer Measurement Techniques", Fifth Annual Workshop on Microprogramming, University of Illinois, SIGMICRO, Sept 25-26, 1972.

[SAA75] Saal, H.J., and Weiss, Z., "An Empirical Study of APL Programs", Computer Languages, Vol 2, pp. 47-59, Pergamon Press, 1977.

[SAL] Salvadori, A., Gordon, J. and Capstick, C., "Static Profile of COBOL Programs", ACM SIGPLAN Notices, Vol 10, No 8, August 1975, pp. 20-33.

[SHU] Shustek, L., "The Internals of the Video Graphics Terminal", Stanford Linear Accelerator Center Report #199, December 1976.

[SLA] Slavinski, Richard T., "Static FORTRAN Analyzer", Rome Air Development Center Technical Report RADC-TR-75-275, November 1975. (Available from NTIS as AD-A019 747.)

[SMI] Smith, Alan, "A Modified Working Set Paging Algorithm", IEEE Transactions on Computers, C-25,9, September 1976, pp. 907-914.

[SNI] Snider, D.R., et al, "Comparison of the Amdahl 470 V/6 and the IBM 370/195 Using Benchmarks", Argonne National Laboratory Report ANL-76-50, March 1976.

[VAN] VanTuyl, W.H., "An Engineering View of Performance, IBM System/370 Model 168", SHARE Computer Measurement and Evaluation Selected Papers, Volume II, p 816-829, August 1973.

[VER] Verstege, P., and Wichmann, B.A., "An Experimental Data Base for Computer Performance Information", National Physical Laboratory Report NAC 62, November 1975.

[WIC70] Wichmann, B.A., "Some Statictics from ALGOL programs", National Physical Laboratory eport CCU11, August 1970, Teddington, Middlesex, England

[WIC73] Wichmann, B.A., "Basic Statement Times for ALGOL 60", National Physical Laboratory Report NAC 42, November 1973, Teddington, Middlesex, England

[WIL] Wilner, W.T., "Design of the B1700", Proc. AFIPS 1972 FJCC, Vol 41, AFIPS Press, Montvale, N.J., pp. 579-586.

[WIN] Winder, R.O., "A Data Base for Computer Performance Evaluation", Computer, March 1973, pp. 25-29.

[WIR] Wirth, N, "The Design and Implementation of Modula", Software - Practice and Experience, Vol 7, No 1, Jan/Feb 1977, pp. 67-84.

[WAN] Wang, Li-Chen, "Palo Alto Tiny Basic", Dr. Dobb's Journal of Computer Calisthentics and Orthodontia, Vol 1, No 6, pps. 12-25, May 1976, People's Computer Center, Menlo Park, CA.

[WOL] Wolin, L., "Procedure Evaluates Computers for Scientific Applications", Computer Design, November 1976, pp. 93-100.

[WOR72] Wortman, D.B., "A Study of Language Directed Machine Design", Ph.D. thesis, Computer Science Dept., Stanford U. 1972.

[WOR76] Wortman, D.B., "A Study of High-Resolution Timing", IEEE Transactions on Software Engineering, June 1976, pp. 135-137.

[YU] Yu, Frank, "Modeling the Write Behavior of Computer Programs", Ph.D. Thesis, Stanford University, 1976.

Appendix A

INSTRUCTION MNEMONICS

This appendix contains a brief explanation of the instruction mnemonics which appear in the text or in tables. For more information, see the appropriate manufacturer's literature.

A.1 IBM 37Ø INSTRUCTIONS

KEY:

Reg	Register-to-register instruction
Unn	Unnormalized
Char	Character (string) data
Compl	Complemented (2's complement)
Float	Single-precision floating point operand
Dfloat	Double-Precision floating point operand
Qfloat	Extended-precision floating point operand
Logical	Logical (unsigned) operand
Dec	Packed decimal operand
Halfword	Halfword (2 byte) operand

A	Add	AD	Add Dfloat
ADR	Add Dfloat Reg	AE	Add Float
AER	Add Float Reg	AH	Add Halfword
AL	Add Logical	ALR	Add Logical Reg
AP	Add Dec	AR	Add Reg
AU	Add Float Unn	AUR	Add Float Unn Reg
AW	Add Dfloat Unn	AWR	Add Dfloat Unn Reg
AXR	Add Qfloat Reg	BAL	Branch And Link
BALR	Branch And Link Reg	BC	Branch Conditional
BCR	Branch Conditional Reg	BCT	Branch and Count
BCTR	Branch and Count Reg	BXH	Branch Index High
BXLE	Branch Index Less/Equal	C	Compare
CD	Compare Dfloat	CDR	Compare Dfloat Reg
CE	Compare Float	CER	Compare Float Reg
CH	Compare Halfword	CL	Compare Logical
CLC	Compare Strings	CLCL	Compare String Long
CLI	Compare Byte	CLM	Compare Under Mask
CLR	Compare Logical Reg	CP	Compare Dec
CR	Compare Reg	CVB	Convert Dec to Binary

CVD	Convert Binary to Dec	D	Divide
DD	Divide Dfloat	DDR	Divide Dfloat Reg
DE	Divide Float	DER	Divide Float Reg
DP	Divide Dec	DR	Divide Reg
ED	Edit Dec to Char	EDMK	Edit/Mark Dec to Char
EX	Execute Instruction	HDR	Halve Dfloat Reg
HER	Halve Float Reg	IC	Insert Char Into Reg
L	Load	LA	Load Address
LCDR	Load Compl Dfloat Reg	LCER	Load Compl Float Reg
LCR	Load Compl Reg	LD	Load Dfloat
LDR	Load Dfloat Reg	LE	Load Float
LER	Load Float Reg	LH	Load Halfword
LM	Load Multiple	LNDR	Load Negative Dfloat Reg
LNER	Load Negative Float Reg	LNR	Load Negative Reg
LPDR	Load Positive Dfloat Reg	LPER	Load Positive Float Reg
LPR	Load Positive Reg	LR	Load Reg
LRDR	Load Rounded Dfloat Reg	LRER	Load Rounded Float Reg
LTDR	Load and Test Dfloat Reg	LTER	Load and Test Float Reg
LTR	Load and Test Reg	M	Multiply
MD	Multiply Dfloat	MDR	Multiply Dfloat Reg
ME	Multiply Float	MER	Multiply Float Reg
MH	Multiply Halfword	MP	Multiply Dec
MR	Multiply Reg	MVC	Move Strings
MVCL	Move Strings Long	MVI	Store Byte Immediate
MVN	Move Dec Numeric	MVO	Move Dec Offset
MVZ	Move Dec Zones	MXD	Multiply Qfloat
MXDR	Multiply Qfloat Reg	MXR	Multiply Qfloat Reg
N	AND	NC	AND Strings
NI	AND Byte Immediate	NR	AND Reg
O	OR	OC	OR Strings
OI	OR Byte Immediate	OR	OR Reg
PACK	Pack Char to Dec	S	Subtract
SD	Subtract Dfloat	SDR	Subtract Dfloat Reg
SE	Subtract Float	SER	Subtract Float Reg
SH	Subtract Halfword	SL	Subtract Logical
SLA	Shift Left Algebraic	SLDA	Shift Left Dbl Algebraic
SLDL	Shift Left Dbl Logical	SLL	Shift Left Logical
SLR	Subtract Logical Reg	SP	Subtract Dec
SPM	SET PROGRAM Mask	SR	Subtract Reg
SRA	Shift Right Algebraic	SRDA	Shift Right Dbl ALGBRAIC
SRDL	Shift Right Dbl Logical	SRL	Shift Right Logical
SRP	Shift Dec Rounded	ST	Store
STC	Store Char From Reg	STCM	Store Chars Masked
STD	Store Dfloat	STE	Store Float
STH	Store Halfword	STM	Store Multiple
SU	Subtract Float Unn	SUR	Subtract Float Unn Reg
SVC	Supervisor Call	SW	Subtract Dfloat Unn
SWR	Subtract Dfloat Unn Reg	SXR	Subtract Qfloat Reg
TM	Test Under Mask	TR	Translate String
TRT	Translate/Test String	TS	Test and Set
UNPK	Unpack Dec to Char	X	Exclusive OR
XC	Exclusive OR String	XI	Exclusive OR Immediate
XR	Exclusive OR Reg	ZAP	Zero and Add Dec

A.2 INTEL 8080 INSTRUCTIONS

KEY:
- R Register operand
- M Memory operand
- I Immediate operand

Instr	Data Length	1st oper	2nd oper	
ADD HL	16	R	R	Add
ANDI	8	R	I	And immediate
CALL				Call (stack return addr)
CMPI N	8	R	I	Compare immediate
DEC R	8	R		Decrement
INC R	8	R		Increment
INC RR	16	R		Increment
IOR R	8	R	R	Inclusive Or
JMP				Jump
JMP CC				Jump conditional
JMP U				Jump unconditional
LD A	8	R	M	Load
LD HL	16	R	M	Load
LD R	8	R	M	Load
LOD	8	R	R	Load
LOD M,R	8	M	R	Store
LOD R,M	8	R	M	Load (indirect addr)
LOD R,R	8	R	R	Load
LODI R	8	R	I	Load immediate
LODI RR	16	R	I	Load immediate
POP RR	16			Pop Stack
PUSH RR	16			Push Stack
RET				Return (pop return addr)
RET C				Conditional Return
ROT	8	R		Rotate accumulator
RST				Restart (a short CALL)
ST A	8	R	M	Store
ST HL	16	R	M	Store
ST R	8	R	M	Store
SUBI	8	R	I	Subtract immediate
XCH	16	R	R	Exchange register pair

A.3 DEC PDP11 INSTRUCTIONS

Single-Operand Instructions

CLR	Clear
COM	1's Complement
INC	Increment
DEC	Decrement
NEG	Negate
TST	Test
ROR	Rotate right
ROL	Rotate left
ASL	Arithmetic shift left
SWAB	Swap bytes

Two-Operand Instructions

MOV	Move (load, store)
CMP	Compare
ADD	Add
SUB	Subtract
BIT	Bit test (And)
BIC	Bit clear
BIS	Bit set (Or)
XOR	Exclusive Or

Others

Bxx	Relative Branches (xx is condition)
JMP	Jump (full address)
JSR	Jump to subroutine
RTS	Return from subroutine
MARK	Mark stack
SOB	Subtract 1, branch not zero

All one- and two-operand instructions except SWAB, ADD, SUB, and XOR may have "B" appended to indicate that the operand length is one byte (8 bits) instead of a word (16 bits).

A.4 PASCAL PCODE INSTRUCTIONS

ADI	Add integer
AND	And
CSP	Call standard Procedure
CUP	Call user procedure
DEC	Decrement address
ENT	Enter block
EQU	Equality test
FJP	False jump
GRT	Greater-than test
INC	Increment address
IND	Indexed fetch
INN	Test set membership
IOR	Inclusive or
IXA	Compute indexed address
LCA	Load address of constant
LDA	Load address
LDC	Load constant
LEQ	Less-than or equal test
LES	Less-than test
LOD	Load contents of address
MOV	Move
MST	Mark stack
NEQ	Not-equal test
NEW	New allocation
NOT	Logical negation
ORD	Ordinal value
RET	Return from block
SBI	Subtract integer
STO	Store at base-level address
STR	Store at address
UJP	Unconditional jump
UNI	Set union
XJP	Indexed jump