DECOMPILATION

by

Gregory L. Hopwood

Technical Report #118

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

March 1978

UNIVERSITY OF CALIFORNIA

Irvine

DECOMPILATION

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

by

Gregory Littell Hopwood

Committee in charge:

Professor Julian Feldman, Chairman

Professor Alfred M. Bork

Professor Fred M. Tonge

1978

The dissertation of Gregory Littell Hopwood is approved,
and it is acceptable in quality and form for
publication on microfilm:

Alfred M. Bork

Fred M. Tonge

[illegible]
Committee Chairman

University of California, Irvine

1978

This dissertation is dedicated to Marsha

Contents

Acknowledgments

When I began this dissertation in November 1972, I had little idea it would be five years before I would write these final words of thanks and acknowledgment to my friends and colleagues. These years have been exciting, challenging and rewarding.

To Julian Feldman and Fred Tonge, my advisors and friends, for ten years of patience, encouragement, and guidance.

To Richard Hamming, whose energy and enthusiasm for science serves as an example for us all.

To Ralph Hollis, physicist, computer scientist, humanist, best friend, and best man.

To my parents, who gave me life and supported my education for many years.

To Francis Eggert, who taught me that science could be my life.

To my brother, Christopher, who knew life is more than science.

To Alfred Bork, for serving on my dissertation committee with good humor and interest.

To David Farber and Martin Kay, for many stimulating conversations in our Irvine-Santa Monica car pool.

To Frank Friedman, Mike Ikezawa, Barry Housel, and

Cliff Hollander, for sharing your decompiler expertise.

To the National Science Foundation, the UCI Graduate Division, and the Computing Facility for financial assistance, and to the taxpayers of the state of California for supporting the University and its endeavors.

To my fellow students at UCI, for thousands of interesting hours -- especially Don Loomis, Larry Rowe, Craig Schwerdt, Jim Hobbs, Alan Bell, Craig Will, Michael Pepper, Dick Burton, John Vittal, Allan Foodym, Frank Heinrich, Ken Larson, Bill Crosby, Bill Earl, Eric Olsen, and Robert Ramos.

To the past and present staff of the Department of Information and Computer Science -- especially Barbara Gray, Rose Gagnon, Phyllis Siegel, Rose Allen, and to Virginia Brakeley of the Computing Facility.

Finally, to Marsha, my wife, to whom this dissertation is dedicated. It could not have been done without you.

Gregory L. Hopwood
February 10, 1978

Vita

October 25, 1945	- Born - San Francisco, California
1967	- B.A., Mathematics, University of California, Irvine
1967-1968	- Member of the Technical Staff, North American Rockwell Corporation, Anaheim, California
1968-1969	- Lecturer, University of California, Irvine
1969-1973	- National Science Foundation Trainee in Information and Computer Science, University of California, Irvine
1972-1975	- Senior Programmer, Distributed Computing System Project (NSF grant GJ-1045)
1975-1976	- Senior Development Engineer, University of California, Irvine
1976-	- Senior Programmer Analyst, Sperry Univac Minicomputer Operations, Irvine, California

Abstract of the Dissertation

DECOMPILATION

by

Gregory Littell Hopwood

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1978

Professor Julian Feldman, Chairman

A decompiler is a software tool which can be used to translate programs written in a low level language into a higher level language for the purposes of understandability, documentation, or transferabilty. This software tool is known as a decompiler because the effect it produces is similar to reversing a compilation process.

The research reported in this dissertation addresses the following questions about decompilation:

- For what classes of source and target languages can the translation be done?
- What are the costs involved (human, machine)?
- How complete a translation can be expected?
- What does the translator look like?

The motivation for this work and a history of decompilation are discussed in the first two chapters.

General decompiler design considerations and the design of the particular experimental decompiler built during this research are presented in the next two chapters. The implementation of the decompiler (using LISP on a DECsystem-10 computer) is described and illustrated by example in Chapter 5. The source programs are minicomputer (Varian Data Machines 620/i) assembler language routines, and the target language is a higher level machine oriented language for the same minicomputer. The next chapter details the results of decompiling two larger programs totalling five thousand lines of assembler language text. The performance of the decompiler is analyzed and compared with data from other research projects. A summary of the results of this research and possible future directions in the area of decompilation are discussed in the last chapter.

The conclusion of the dissertation is that decompilation can be a useful aid in the process of evolutionary program improvement. This research should prove helpful to others contemplating the construction or use of decompilation tools for their own research or for production purposes.

Chapter 1

MOTIVATION

INTRODUCTION

Since the mid 1950's much of the effort in the field of programming languages has been directed toward the definition, implementation and use of a class of translators called compilers. These translators accept as input a program written in a particular "high level" language, and they produce an assembler or machine language version of the program. Jean Sammet (1969, 1972) has catalogued over one hundred such languages where each language has at least one compiler and perhaps scores of different compilers, as do FORTRAN and COBOL.

The reasons for the growth of the popularity of high level languages are manifold. They include ease of coding, self-documentation, program transferability, ease of debugging, and problem oriented syntax. Few high level languages or systems for writing programs in high level languages exhibit all of these attributes, but the use of each language presents some advantage to its users over programming in a lower level language such as assembler code.

CONTINUED USE OF ASSEMBLER LANGUAGE

An interesting question arises to challenge those who might give unqualified support to the concept that high level language systems are such an advance over old fashioned assembler language coding -- Why is so much programming still being done in assembler language? Philippakis (1973, 1977), in a survey of institutions which do business data processing, reports that the amount of effort devoted to assembler language programming is exceeded only by COBOL programming.

In the area of systems programming, typically the development of very large operating systems or system utilities such as editors, debuggers and translators, we see an extensive use of assembler language in preference to higher level languages. In the past several years there have been some attempts to move to higher level languages for system implementation. Languages used by manufacturers for system implementation include PL/I (the principal implementation language of the MULTICS system; also used by IBM), ALGOL (Burroughs), PL/M (Intel), FORTRAN (CDC, Prime), PL/S (IBM), DGL (Data General), BLISS (DEC), and SPL (Hewlett-Packard). Most minicomputer manufacturers, however, still write most systems routines in assembler language to the general exclusion of higher level languages.

The Reasons Why

There are four major reasons for continued use of assembler language:

1. Coding efficiency -- Many compilers do not create code as efficient in terms of speed of execution or storage size as a good human programmer coding in assembler language.
2. No appropriate language or language compiler available -- Many languages are not suitable for systems programming because they do not allow the programmer to access low level system elements such as interrupts, channels, status words, etc. They may also not provide the ability for the programmer to describe and efficiently manipulate data structures common to system tasks. Market lead time for the product may not allow for the definition and/or creation of such a compiler.
3. Costs of compilation -- The compiler time costs may be excessive in terms of machine time or real time. In the case of minicomputer (or microcomputer) manufacturers, there may be no machine on which the high level language programs may compile because the minicomputer is too small to support the compiler itself, no cross-compilers exist, and the cost of developing and using one is excessive.
4. Compatibility -- Much of the system software is

already coded in assembler language and new releases and additions must be compatible.

Efficiency Considerations

The fact that there are high level languages in use for systems programming work indicates that the above objections to their uses are not all pervasive and unconquerable. Yet, we must face the fact that assembler language programming is extensively used and will continue to be so for quite a few years to come because the most serious objection mentioned, coding efficiency (or lack of it), affects the saleability of a manufacturer's product much more so than the system's maintainability or documentation. Coding efficiency directly affects system performance under benchmark tests, one of the most widely used comparators for computer selection (see Timmreck 1973).

Optimizing Compilers

The automatic production of efficient code by software means is possible but expensive. It can be done with clever compilers and optimizers. These tools cost money to build, and more importantly in some cases, time to build. Most computer manufacturers and users have not decided to spend their resources in that direction. One quite well-known optimizing compiler has been built by IBM for the FORTRAN language (see Lowry and Medlock 1969). Although not built for systems programming tasks, this optimizing compiler does represent an attempt to create efficient code from a higher level language. The optimizing pass of the compiler represents such a large amount of overhead relative to a simple compilation that its use is an option that is normally avoided by a user who is debugging his program. After the program is debugged then the optimizing pass is performed.

Future Trends

The use of instruction set enhancement via writable control store microprogramming techniques can greatly increase the efficiency of high level language programs. In essence, the firmware interpreter is made to execute the operations of the high level virtual machine in an efficient manner. For example, the FORTRAN DO loop mechanism, or the stack operations required by an ALGOL program, can be optimized with special firmware instructions. Microprogramming to create efficient execution environments for the code produced by high level language compilers promises to help close the efficiency gap so often cited as a reason for continued use of assembler language. Minicomputer manufacturers have pioneered this technique of using firmware "accelerator" routines in combination with writable control store micro-engines. Application of this principle for the emulation of complete high level virtual machines will become common place.

The future course of language use is clear. Low level languages are on their way out but the need for writing and maintaining programs in assembler language will probably remain with us for the next decade. Is there a tool which can help us make the transition to the predominate use of high level languages for system implementation and help us move our present huge inventory of valuable low level language software to our future machines?

AIDS TO UNDERSTANDING

Written Specifications

Programmers and software managers have long recognized the pitfalls of assembler language programming. This has led to various management dicta on how programs should be written and documented. Management calls for programmers and analysts to write specifications in natural language. These specifications are produced to describe several levels of detail, functional, external and internal, yet the prime source of information about a program continues to be the program text itself. The is the final authority about what the program does (rather than what it should be doing).

Program Text

Comments are included by good programmers in the text of the source program to give an alternate and more general description about what the machine instructions are supposed to be doing. Several problems may occur when trying to rely on the program text for information, however. The program text may not actually represent the current object version being used on the machine because of patches, programmer failure to keep the most up-to-date source version of the program, or file system problems. In addition, the comments on the listing may give the reader the wrong impression about what the program is doing.

Flowcharts

An additional form of program representation, the flowchart, has evolved to provide some additional aid to people who must read programs rather than just write them. Flowcharts describing the top level interaction of program modules are often useful, but they only provide a program overview and not the detailed informaion that is needed to correct or improve a program. Flowcharts written at lower levels tend to degenerate into so many pages and off-page connectors that comprehension of the flow of control is almost impossible. Unfortunately, most automatic flowcharters for assembler language programs create these low level descriptions. These flowcharters do not synthesize higher level structures such as expression evaluations or loops into a form that can fit on a few pages of a printout. (Climenson (1973) has described a method for reducing the number of off-page connectors in automatically generated flowcharts.)

Since written descriptions, program comments, and flowcharts are often inadequate aids to program understanding, is there a tool which can help us better understand low level language programs?

THE DECOMPILER

This dissertation is about a software tool which can be used to translate programs written in a low level language into a higher level language for the purposes of understandability, documentation, or transferabilty. This software tool is known as a <u>decompiler</u> because the effect it produces is similar to reversing a compilation process. As an example, Figure 1-A presents a segment of assembler language code for a single accumulator machine and a corresponding segment of code in a higher level language -- the possible output of a decompiler given the assembler code as input. Most programmers would agree that the higher level program segment is easier to read and more clearly indicates what the program is doing.

DISSERTATION OBJECTIVES

This dissertation will attempt to answer the following questions about decompilation:

- o For what classes of source and target languages can the translation be done?
- o What are the costs involved (human, machine)?
- o How complete a translation can be expected?

o What does the translator look like?

SUMMARY OF FOLLOWING CHAPTERS

Chapter 2 discusses what has already been done in the area of decompilation by other groups or individuals. Chapter 3 discusses general decompiler design considerations. Chapter 4 describes the design of a decompiler as an experimental vehicle for the exploration of various decompilation techniques. Chapter 5 describes the implementation of the design presented in Chapter 4 and illustrates the workings of the decompiler by passing a small example through each step of the system. Chapter 6 presents some results of decompiling two larger production programs and compares the performance of the decompiler with other similar systems. Chapter 7 summarizes the results of this study and discusses possible future research directions in the area of decompilation.

```
        Source                        Target

        LDA     X                IF X+10 # Y THEN
        ADD     =10
        SUB     Y
        JAZ     L1
        JMPM    SUBR                CALL SUBR
        JMP     L2
L1:     LDA     J                   ELSE J := J+2 ;
        ADD     =2
        STA     J
L2:     LDX     I                B[I+1] := C[I] - J ;
        LDA     C(X)
        SUB     J
        IXR
        STA     B(X)
        HLT                      HALT;
```

Figure 1-A. Sample decompiler input and output.

Chapter 2

REPROGRAMMING TECHNIQUES AND DECOMPILATION SYSTEMS

INTRODUCTION

<u>Reprogramming</u> is the general term under which we will discuss the problem of transferring a program written for one machine to another machine. This is also known as the <u>transferability</u> or <u>portability</u> problem. A program which can easily and automatically be transferred from one machine (source) to another (target) is called <u>machine independent</u>. As we shall see, machine independence is a goal which is rarely completely attained. Because decompilers were initially developed as a reprogramming tool, we include in this chapter a discussion of other reprogramming techniques along with a review of past decompilation efforts.

SIMULATION

Simulation is the most general reprogramming technique. The source program is interpreted by the target (or more descriptively, the host) machine via a <u>software</u> system which examines each instruction of the source program and then executes a set of host instructions which will produce the desired effect. The simulator usually consists of a software representation of the control and functional components of the target machine. A pseudo location counter, memory, and state information are kept by the program. The simulation is usually done at the bit level, where every bit of information which is in the source machine appears in the host machine. A functional simulation, on the other hand, is concerned with producing the same effect as the source program running on the source machine, but every bit of information may not be faithfully represented.

Simulation has several advantages in the context of a reprogramming effort:

- A simulator is easily instrumented so that the functions of the simulator can be easily traced.
- The cost of building a simulator for a machine may be quite low compared to a hardware implementation (depending on the match between the source and host machines), particularly if the entire source machine is not modelled.
- If the source program needs to be run only a few times on the host machine, the cost of simulating the source program may be negligible, especially if a simulator already exists.
- The source machine need not be a "real" machine. Paper machine designs can be tested, at least at

the functional level, before they are built. Knuth's MIX machine (Knuth 1968) is an example of a paper machine that is used for pedagogical purposes in many colleges and universities.

Disadvantages of simulation can be listed:

- o Since every hardware instruction of the source machine is interpreted through a software routine, the ratio of host machine cycles to source machine cycles can be very high, typically 50 to 1 or greater. This can lead to intolerable machine and real time costs for production programs.
- o Hardware characteristics, such as time dependencies and input/output devices, can often not be simulated and may cause some programs to be non-transferable even by simulation.

EMULATION

Emulation used to be understood as the interpretation of the source machine code by a host which is hardwired to perform the instructions of the source machine in almost a one-for-one simulation. The introduction of microprogrammed machines with writable control stores has blurred the distinction between simulation and emulation, but the distinguishing characteristic is the presence in an emulator of a <u>hardware</u> capability in the host machine to interpret the source machine code without a significant (order of magnitude) degradation in performance.

Emulation is a capability sometimes offered by computer manufacturers in order to attract customers away from the competition with the promise of program compatibility. Emulation is also used to provide a way for current customers to move software from obsolete machines to more modern ones in a new product line. The IBM 360/65 emulation of the older 7090 series machines was an example of the latter case. The native instruction set of the 360 was incompatible with the 7090 but the emulation feature of the model 65 enabled 7090 customers to make the transition to third generation hardware without a massive reprogramming task. The software could be gradually translated or retired from service without the necessity of running both real machines simultaneously.

AUTOMATIC TRANSLATION

An automatic reprogramming system translates code written for the source machine into code which will execute on the target machine, without any manual corrections being required. This approach to portability works well as long as the source and target machine architectures are similar, that is, all operations which may be invoked on the source machine have a corresponding operation or set of operations which will cause the same net effect on the target machine. In other words, the source machine operations are covered by the host machine.

Most translators in existence today are examples of this class of reprogramming aid. For instance, a FORTRAN compiler translates code written for a hypothetical FORTRAN machine into a set of instructions for a real machine. The definition of the FORTRAN machine has been standardized in an attempt to provide the ability to transport FORTRAN programs from one host to another. The computer industry has been partially successful in adhering to such standards for FORTRAN, but the temptation for a manufacturer to "improve" the definition of the FORTRAN machine is almost overwhelming, so that many super-sets of the standard FORTRAN exist. This leads to incompatibilites between manufacturers. The desire of FORTRAN users to write their programs in a machine dependent manner for the sake of efficiency or expediency leads to FORTRAN programs which cannot be automatically translated. (In the above discussion, the reader may substitute the name of almost any higher level language for "FORTRAN.")

Unfortunately, the programs which are prime candidates for transportation to different computers are often those which exhibit the obstacles to such a move. Because of their production nature they were programmed to run efficiently, and hence many machine dependent features were used to save space and/or execution time. Automatic translation of these programs becomes an unattainable goal.

SEMI-AUTOMATIC AND MANUAL TRANSLATION

It is often the case that a program written in one dialect of FORTRAN or other high level language can be translated almost completely and automatically by a compiler for another dialect. Usually there remain small but annoying changes which must be manually made to the program before it will compile in toto or execute properly. The advantage of these semi-automatic systems is that they do most of the work.

The difficulty of applying semi-automatic translation techniques to transport low level language programs from one machine to another is proportional to the differences between the source and target machine architectures and how often those differences are important to the proper execution (meaning) of the program. For example, the source machine may use a signed magnitude representation of integers, while the target machine uses two's complement. Whether the ORing of a one into the sign bit of a data item is intended to do sign conversion or is just a logical bit modification to be used as a flag, determines how the instruction will be translated.

When a semi-automatic translator seems useless because of the incompatibilities between the source and target machines, the only avenue left is manual translation. In that case, a human being determines the algorithm the source program represents and translates the algorithm (rather than the program). The algorithmic representation may be considered as some abstract representation which is independent of any real machine architecture.

A SURVEY OF DECOMPILATION SYSTEMS

Decompilers fall into the class of semi-automatic or automatic reprogramming aids. Figure 2-A illustrates how the decompilation process can help to achieve program transferability. A decompiler is distinguished from the other reprogramming aids mentioned earlier in that its modus operandi involves the translation of low level language programs into ones expressed in a higher level language. This higher level language is then recompiled to run on the target machine. Housel (1973) has described the early decompilation efforts preceding his work. Those earlier efforts will be mentioned here in less detail. Housel's work and others he has not discussed will be presented in more detail.

Halstead

The origin of the term "decompiler" has been attributed to the 1960-1962 decompiling project at the Navy Electronics Laboratory where a series of translators were developed by Halstead, Englander, and Donnelly to aid in the conversion of software for CDC, UNIVAC, and IBM machines to the NELIAC language. This work is reported by Halstead (1962, 1967, 1970). This effort was successful in translating up to 98%

of some programs, but manual editing of the results was still often necessary. (It is interesting to note that the same M. H. Halstead, in 1971 at Purdue University, became a co-principal investigator of an NSF funded "Inverse Compiling Study" (Grant GJ 31572) and has recently renewed the academic interest in decompiling through this project which has directly or indirectly produced several Ph.D. dissertations in the area.)

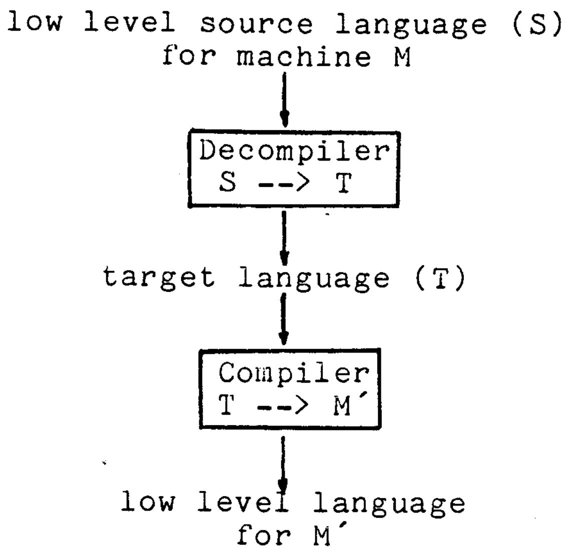


Figure 2-A. Decompilation for program transferability from machine M to machine M´.

Sassaman

In the case of the work of Sassaman (1966) at TRW, the source language was IBM 7090-7094 assembler language and the target language was FORTRAN. The motivation behind this translation effort was to aid in the conversion of their second generation software to a new third generation computer. It appears to have been successful in its goal of translating scientific application programs by performing most of the clerical drudge work of translation and in the initial and final phase providing capability for human editing and guidance if necessary. Unfortunately, this work was never fully reported and the five page article referenced here (Sassaman 1966) does not present any non-simple examples or statistics to indicate in detail the structures, algorithms, or performace of the system.

IBM -- ACCAP

Housel reports on the IBM ACCAP (IBM 1967) "Autocoder to COBOL Conversion Aid Program" which has as its source machine the decimal, variable word length 1400 series of computers. This was another aid to relieve the shock of the second/third generation computer transition. According to Housel, ACCAP created inefficient, often one-to-one translations which were on the average 2.1 times larger than the source in terms of core storage usage. This type of translation made human optimization of the output desirable, if not necessary.

The use of this type of translation can be justified on several grounds, despite its inefficiencies. If the target program will only have a short lifetime of use, inefficiency can be tolerated. The mechanical translation phase frees humans from the laborious task of total translation and often human time is much scarcer than machine time. The inefficiencies introduced may not be significant with regard to the resources available. For example, if the output of a decompiler is within the maximum bounds of space and time for the system on which the target program is to execute, then the fact that the program may be three times larger or two times slower than the original may be no real concern.

<u>Barbe</u> -- PILER

The PILER effort reported by Barbe (1969, 1974) is an example of a system designed to handle a large class of input languages and target languages. The structure of PILER is shown in Figure 2-B. The input language is decoupled from the system by an <u>interpreter</u> which converts the machine language to a standard "micro-form" format. This format is essentially a pseudomachine language for compact representation of machine language instructions. The <u>analyzer</u> reads micro-form text and converses via a flow-chart language with a human operator. It builds a program in an intermediate language which is input to a phase called the <u>converter</u>. The converter decouples the decompile analysis phase from the target language. It accepts the intermediate language output by the analyzer and translates that to the compiler language.

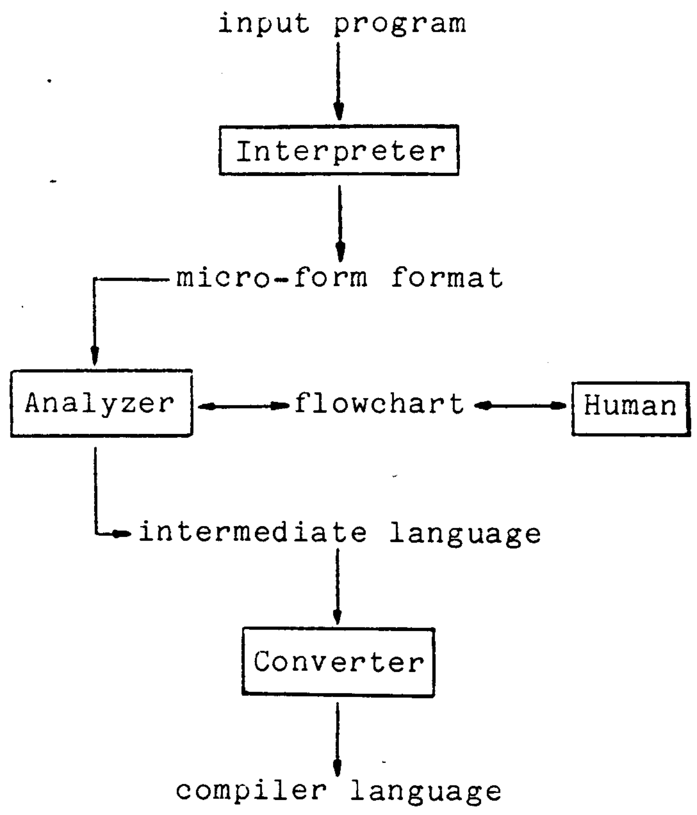


Figure 2-B. The structure of PILER.

The design of PILER is of special interest as a prototype for future decompilers which are built to handle more than one source language/target language pair.

It is a classic example of a three-stage translator where the first stage decouples the input from the inner workings of the system, the central portion does all the work in a standard format, and the final stage translates the standard format into the output language. Some meta-compilers provide another example of this kind of organization. The first stage reads syntax tables and parses the input language into an intermediate (usually a parse tree) format. This intermediate language is analyzed and an assembler language is output which is then translated by an assembler to the target machine. The use of a standard assembler for the final translation frees the meta-compiler from the chore of creating actual machine code.

The interesting questions with regard to the PILER type of translator organization are:

- o What is the effort required to match the interpreter to the input? Do we write some subroutines and change some tables? Is the structure of the interpreter itself unchanged?
- o What class of input languages can the interpreter be expected to accept?
- o How much do we pay for generality in the execution speed of the translator?
- o Is the analysis phase really independent of the input language and output language? If not, what changes must be made to accomodate different source/target pairs? How many <u>ad hoc</u> data structures, or special case code segments appear in the analyzer? This question really asks whether the analyzer is <u>completely</u> decoupled from the source and target languages.
- o Can the output section be easily accomodated to different target languages? What classes or

characteristics of the output language are critical?

PILER's micro-form code seems its most obvious weak point. This low level 36-bit compressed format is an attempt to reach a lowest common denominator for machine languages. The tight coding of the micro-form text does not leave the PILER design adaptable to changing requirements imposed by new source and target language pairs.

Hollander

In his Ph.D. dissertation, Hollander (1973) describes a decompiler designed around a formal syntax-oriented metalanguage. An implementation of a decompiler to translate a subset of IBM 360 assembler language to ALGOL is discussed.

Hollander's decompiler model was meant to be applicable to a large number of source/target language pairs. His design is significantly different from any other decompiler system reported in the literature. Figure 2-C illustrates the five phases of Hollander's decompilation process. However, it is the implementation of each of these phases as an interpreter of sets of meta-rules which distinguishes his work.

(The techniques of using meta-rules (or programs) to separate the workings of an interpreter (or machine) from any particular application of the interpreter was an early fundamental contribution of computer science to information processing. The most common example of these techniques is seen in the stored program computer. These techniques have also been applied to the design of translators and are summarized in Feldman and Gries (1968).)

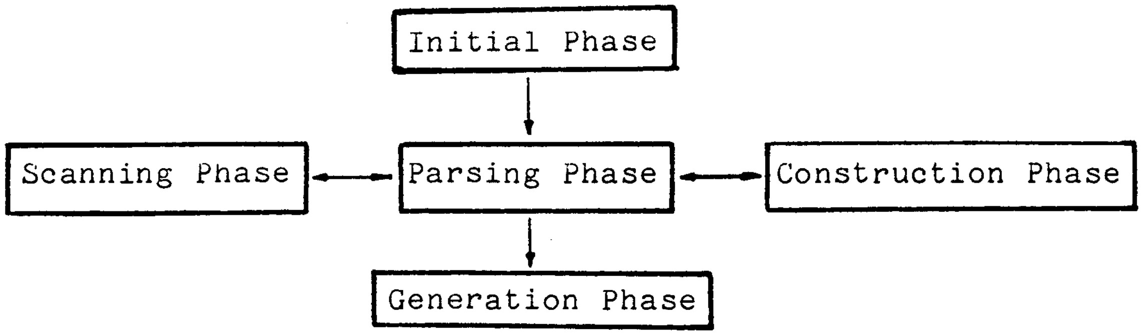


Figure 2-C. Phases of Hollander´s decompilation model.

Hollander, seeing the inherent commonality of processes between the operations of a compiler and a decompiler, has adapted some of the techniques used in contructing translators to the construction of his decompiler. On the surface, this approach might seem to lead to a large pay-off in the adaptation of well-developed and analyzed methods to a new use. While the syntax-directed methods advocated by Hollander are very useful in decompiling certain classes of low level programs, there is a fundamental weakness in this design leading from the basic differences in the nature of compilation versus decompilation.

Syntax-directed translation is essentially a pattern-matching operation where the <u>static</u> structure of the text to be translated is examined through a "window" which is moved over the text. When a pattern is recognized, the text matched is replaced by a token symbol, some code may be generated, or an error is signalled. In any case, the syntax rules specify a structure (pattern) which the programmer who produced the input text must have followed in order to produce a legal program. Assembler language is a _weakly_ structured form for expressing computer algorithms because the number of patterns which the programmer can produce to create a given effect is not constrained by many syntactical rules. Therefore, the meaning (semantics) or intent of the sequence of assembler language statements (and its decompiled form) may be less than clear.

Decompilers which utilize a syntax-directed approach can only hope to translate static patterns or structures which appear in the low level program. Hollander's decompiler would work very well on translating the output of a compiler back to the original source language (as he has apparently done in some experiments with an ALGOL-W decompiler) because for every source statement in the higher level language, the exact pattern of object level instructions is known (assuming no optimization has been applied). These patterns could be used in the syntax rules of the decompiler to reverse the translation. As Hollander states in the conclusion to his dissertation (p. 151):

> "The principal limitation of the decompilation scheme developed here is the inherent difficulty of mapping a

weakly structured (source) language into a more highly structured (target) language. The disparity between degrees of structure of the two languages introduces difficulties into both the analysis and synthesis aspects of decompilation."

So many special cases of unexpected statement sequences occur in most assembler language programs that they should be handled by a global analysis phase which constructs the control flow graph of the programs -- looking for patterns in the <u>control graph</u> rather than in the assembler language program itself.

One of the great benefits of structured programming is that the control flow of a program is represented clearly in the syntactic structure of the program text. In "GOTO-less" programming, a control pattern has only one of a very few counterparts in the higher level language. This one-to-one mapping is what makes the understanding of well-structured programs easier. The pieces of an unstructured program text represent many possible primitive control paths when viewed through the small window of a syntax scanner. The whole pattern of control flow may not be recognized because of redundant or misleading statements which appear in the text. These would be filtered out if a control graph were used to recognize structure.

One way to relieve syntax-directed decompilers of this problem of control flow recognition is to have a preparatory phase which "re-writes" the assembler language program by putting it into a standard form for the parse rules to recognize. This phase would normalize the input to the parsing phase and reduce the number of special case patterns necessary to recognize each type of target language structure.

In summary, Hollander's decompiler is important in the evolution of this type of translator, since it represents the first reported attempt to apply the metalinguistic techniques of compiler writing to the task of creating decompilers.

Housel

The Ph.D. dissertation of Housel (1973) represents an approach to decompilation similar to that of Barbe's PILER and very much different from that of Hollander. Housel has attacked the weak point of PILER, the micro-form text, and, in addition, presented a major contribution to the exposition of the nature of decompilation in general and the process-directed (as opposed to syntax- or pattern-directed) approach in particular.

The thrust of all of the decompilation efforts discussed has been toward the use of decompilers for program transferability. Housel's decompiler is also oriented toward that goal. Housel created an experimental decompiler which translated Knuth's MIXAL assembler language (Knuth 1968) to PL/I. The major phases of Housel's decompiler are shown in Figure 2-D.

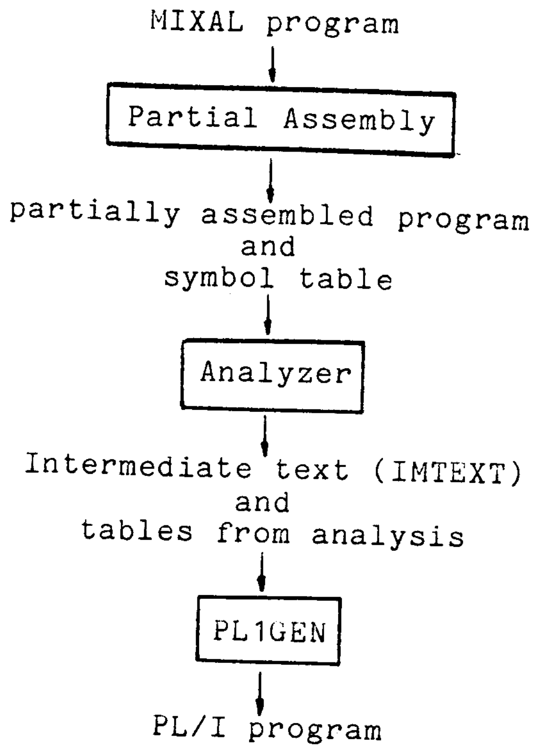


Figure 2-D. Housel´s decompiler model.

Note that the input language is not decoupled from the analyzer as in the PILER model. However, from Housel´s description of his decompiler it appears that most of the work is done on the IMTEXT version of the input and thus provides the buffering level necessary for some machine independence.

IMTEXT is a general "assembler language" which specifies data movement between storage elements and the operations upon them. The format is flexible and adaptable. (In the course of my early research I also independently developed a similar approach to the encoding of program information in a source-target machine independent format.)

The economy of storage of information shown in PILER micro-code is traded for generality and ease of adaptation to new requirements.

Friedman

The Ph.D. dissertation of Friedman (1974) represents an effort to transport minicomputer operating systems code written in assembler language from one machine to another within the same architectural class by modifying Housel's decompiler system (Housel 1973). Friedman follows the prototype model of the transfer process illustrated in Figure 2-A. The target language of the decompiler is FRECL, a systems implementation language designed by Friedman and oriented toward the class of machines (M), generally known as minicomputers. Friedman built a FRECL compiler to translate the output of the decompiler to the machine language of the target machine, a Microdata 1621 computer.

In his first experiment using the decompiler, Friedman converted portions of the IBM 1130 Disk Monitor System to FRECL and reported on the problems involved in the translator process. In his second experiment, the operating system of the Microdata 1621 machine was translated to FRECL and then compiled back into 1621 machine code. He reported on the results of the various stages of this process.

Since the work described by Friedman in his dissertation closely resembles that conducted in my own research, a comparative study of the results of these two projects will be presented in Chapter 6.

<u>de</u> <u>Balbine</u> -- The "Structuring Engine"

A different facet of the uses of decompilation is represented by the work done by Guy de Balbine (1975). The "Structuring Engine" (about 30,000 lines of PL/I code) restructures FORTRAN programs for IBM, CDC, UNIVAC, and Honeywell computers into a super-set language called S-FORTRAN. S-FORTRAN provides the structured coding constructs which are missing from regular FORTRAN. The translation process may rearrange or rewrite the program in order to meet constraints imposed by structured programming techniques; however, the resultant program is functionally identical to the original with respect to its execution (ignoring possible timing considerations).

The S-FORTRAN version of a FORTRAN program is used as an aid to understanding the original program. In addition, the new S-FORTRAN version of the program text can become the new working source for the program since an S-FORTRAN to FORTRAN translator provides the user with the capability of compiling the S-FORTRAN text back into FORTRAN.

This technique of using decompilation to achieve a more understandable representation of the source program, while maintaining the ability to translate the resultant text into executable code, is exactly the theme of the research described in this dissertation. The "Structuring Engine" is aimed at a relatively narrow range of source/target program language pairs, namely FORTRAN and S-FORTRAN, but the volume of code written in FORTRAN is so substantial that the usefulness of such a translator is obvious. The emergence of a commercial product like this confirms my belief that such translation systems are viable tools which will be accepted by the software community.

Ultrasystems

In the Spring of 1974, I consulted on a decompilation project for Ultrasystems, Inc., (Newport Beach, California). The purpose of this project was to create a decompiler to be used as a documentation tool for the Trident submarine fire control software system. The decompiler was to translate Trident assembler language to a higher level programming language known as THLL (Trident Higher Level Language). The Trident computer is a 32-bit word machine whose instruction set can briefly be described as a super-set of the IBM 360/370 order code including many "hardwired" trigonometric and stack instructions not found in many commercial computers.

This decompilation design effort was unique in several respects:

1. The source language and its host computer were not finalized in their designs. No production versions

of the machine existed.

2. The target language (THLL) was in the design stage.
3. The potential candidate programs for decompilation would be those least amenable to decompilation, since the bulk of programming for Trident would be done in THLL and only those programs very critical in time, space, or other operational constraints would be programmed in assembler language. On the other hand, these programs, due to their critical nature, must be understood in detail by the software staff engaged in the maintenance of the software system. The decompiler was to be one of the aids to such understanding.
4. The source programs would be characterized by a heavy use of macros which would not be symbolically expanded.

The Trident control computer exhibited many architectural traits which would make a comprehensive decompiler a formidable program, for example:

a. condition codes with many different meanings assigned to the states depending on the instruction being executed,
b. byte, half-word, double-word, integer, and floating data items,
c. comprehensive stack instructions,
d. variable field instructions,

e. built-in trigonometric and matrix operations, and

f. built-in iterative operations.

The flow of information through the Ultrasystems decompiler is shown in Figure 2-E. The listing output (AL) from the Trident assembler system is input to a preprocessor. Data areas are recognized (by the pseudo-operations declaring such areas) and passed to the data declaration section of the decompiler. The program code (AL*) is then converted to an intermediate form (IM). Arithmetic and logical expressions are built up from subexpressions of single assembler language statements (IM*) and the higher level language (THLL) is output along with the data declarations and types gathered earlier.

Since my task for the nine days I consulted with Ultrasystems was to provide the design of the system, the reader will note the similarities of this design to that discussed later in Chapter 4. The Trident source machine (and its assembler language programs) was much more complex than the source machine used in my own research. This fact required the creation of special algorithms and information tables specific to the Trident computer. Of particular concern was the nature of the side-effects of instructions influencing the condition code settings. See Chapter 3 for a discussion of this problem.

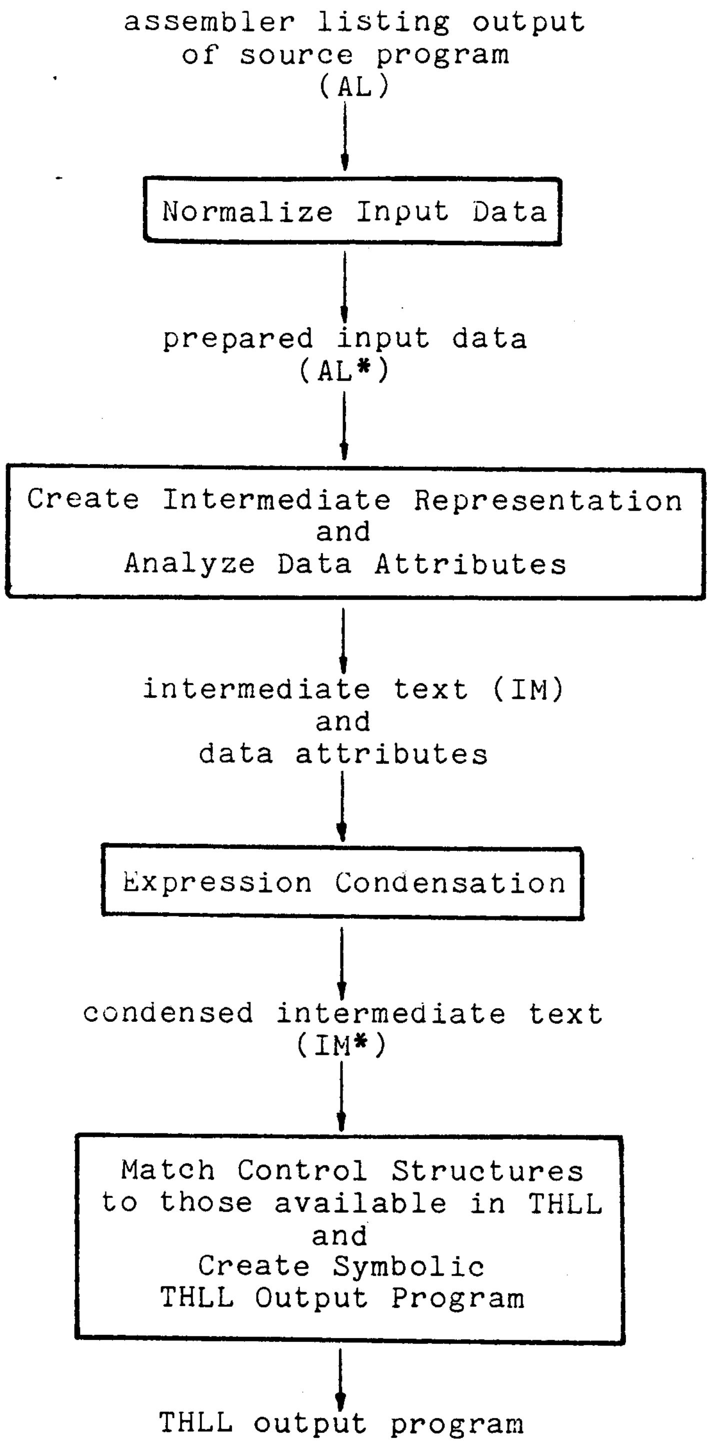

assembler listing output of source program (AL)
Normalize Input Data
prepared input data (AL*)
Create Intermediate Representation and Analyze Data Attributes
intermediate text (IM) and data attributes
Expression Condensation
condensed intermediate text (IM*)
Match Control Structures to those available in THLL and Create Symbolic THLL Output Program
THLL output program

Figure 2-E. Ultrasystems decompiler model.

The Ultrasystems decompiler was not required to create executable code in THLL. The output was to be used for documentation purposes. The success of the effort must be gauged according to that criterion. Because of time constraints, the expression condensation portion of the decompiler was not coded and this affects the readability of the output since all expressions are of the form "A" or "A op B" where A and B are primitive names. The Ultrasystems decompiler appears to be a good first attempt at creating this type of translator for a very complex machine. These pragmatic solutions to many of the problems in decompiling Trident programs (problems which do not occur in many less powerful computers) deserve an examination by any one considering undertaking a similar task.

The Ultrasystems decompiler was written in AED and executes on a CDC 6700 computer. Documents describing the decompiler are listed under Ultrasystems (1974).

Ikezawa -- AMPIC

An interesting use of a decompiler-like system is shown in the work of Michael Ikezawa and others at Logicon, Inc. (Ikezawa and Kayfes 1975, Ikezawa 1977). AMPIC is a system written in SNOBOL which translates assembler language to FORTRAN and interacts with a user to analyze program paths and conditions required to take those paths through the program. It does this by symbolically executing the program. Many of the processes necessary to a decompiler are present in AMPIC. Other parts of AMPIC are used in verifying the existence of certain conditions or the truth of predicates introduced by the user. The main purpose of AMPIC is the validation of low level language programs, such as are used in small tactical computers. AMPIC has also been used successfully to verify a FORTRAN compilation of a larger program. This was done by running AMPIC on both the source and object programs and automatically comparing the AMPIC outputs (Ikezawa 1976).

In the case of AMPIC, we see a translator with many of the features of a decompiler being used for the purposes of analyzing a program in a quantitative and qualitative manner. AMPIC enables a user to examine a program in new and interesting ways: the program is structured, presented as a flow diagram, paths can be traversed symbolically and the low level program can be represented in a higher level language. This blend of decompilation and verification techniques is an important contribution to the technology of program analysis.

Proprietary Commercial Decompilers

The importance of the work reviewed in this chapter under the title "Decompilers" is that each was an attempt to explicate in the public literature information about the decompilation process so that the "wheel" will not have to be reinvented each time around. The contribution of proprietary commercial decompilers to the state of the art has been minimal because of the lack of publication in the open literature. This secretiveness is understandable and justified by financial considerations. Contributions to computer science, however, are only recognized as such when someone besides a small group of insiders knows about them. If there have been great advances in the technology of decompilation by commercial interests, they will have to go unknown as long as they are unpublished. We can only guess at what goes on inside the commercial decompiler by reading the advertisements in the trade magazines and newspapers.

SUMMARY

The viability of decompilers -- their usefulness compared with other methods of reprogramming -- has been demonstrated by their successful application in diverse environments. Many decompiler applications have been ad hoc and unpublished. Others like those reported here, have been attempts to achieve more generality and to extend the decompiler technology. This dissertation builds upon and augments this work.

Chapter 3

DECOMPILER DESIGN CONSIDERATIONS

INTRODUCTION

The purpose of this chapter is to provide a discussion of the questions to be considered when designing a decompiler. We will examine these questions under the following topics

1. preparation of the source program,
2. loading of the source program,
3. internal representation of data in the decompiler,
4. transformations on the internal representation,
5. translation to the target language, and
6. other considerations.

We begin by defining a <u>decompiler</u> to be a translator which accepts as its input some program (PL) written in a low level language (L) and produces, possibly with the aid of manual intervention and other external information, a representation (PH) of PL in some higher level language (H). The translation may be done to help a human to understand PL and/or for documentation purposes. In that case, PH need not be a program that runs on any machine. Its sole function may be to represent PL in a form more easily interpreted by human beings. The translation may also be done for the purpose of transferring PL to a machine which will be able to interpret PH in an equivalent manner. The two goals of decompilation, understandability and transferability, are distinguished by the interpreter for which the output is intended.

PREPARATION OF THE INPUT

What will be the form and content of the input to the decompiler? The primary choices are machine language, symbolic text, or a mixture of both with or without additional information about the program which might prove useful to the decompiler.

Machine Language

A compact representation for PL is its machine language version. The decompiler could have a pseudo memory much like a simulator might have that would contain the PL machine language program bit-for-bit as it would appear in the memory of the machine for which it was written. A four thousand word 16-bit/word machine language program would represent an assembler language program whose listing was about seventy pages -- a fairly good sized program. This program could be accomodated in machine language form in 64 thousand bits, or 2K words on a 32-bit machine. The use of this method of representing PL would alleviate any need to worry about using auxiliary memory such as disk for the program text. It allows for the accessing of any portion of the program without concern about whether that part is or is not in the main store of the decompiler. The decompiler does not have to be an assembler if a machine representation is adopted.

The use of a machine language representation has some disadvantages, however. When a program is assembled and linked, all of the symbolic information which was present in the assembler language source is translated into a machine representation. Often this translation is many-to-one. That is, several symbols in the assembler program may have had the same assembly time value, but for mnemonic purposes the names were different. This distinction is obliterated by the assembly process. The fact that certain values which appear in the machine language program are absolute or relocatable is also lost after a link step in the translation. Where does the symbolic information come from? We can adopt a strategy similiar to disassemblers used with dynamic debuggers, and keep a symbol table around without the program source. This symbol table can be generated from the output of an assembler or by a human.

If a symbolic source of PL is not available the decompiler has to generate its own symbols, for example a serialization using a root prefix. The declaration and initialization of data areas of PH can be deduced from the core image of PL and the patterns of variable usage and instruction flow or from a symbolic source.

The decompiler can perform many functions without a symbolic source. For example, we may wish to know whether a certain piece of code is in a loop, and if so, what is the terminating condition of the loop. Such information can be extracted from a machine language representation of the program but for the purposes of human readability, a human generated symbol table would be helpful.

Symbolic Information

This dissertation treats specifically the problem of decompiling programs written in assembler or machine language. A machine language and assembler language representation of PL describe the same set of basic instructions for the machine for which they are written. The main difference between them is their symbolic or mnemonic content.

The symbolic information contained in an assembler language program can be passed through the translation steps of the decompiler and appear in the output. Labels, comments and symbol usage can be preserved faithfully. Since most programs considered as candidates for decompilation have a symbolic source form, this information should be used.

A compelling reason to use the symbolic text of PL is that form may contain information which yields valuable clues about the program structure or data attributes which are obscured by the assembly process. Examples of this kind of information are

- a subroutine ENTRY statement indicating an entry point to a subroutine;
- a RETN statement indicating a subroutine return which may be actually implemented as a common indexed transfer;
- a DATA statement indicating the initialization of a variable in symbolic terms which will help differentiate the data value from just another bit pattern;
- storage reservation statements indicating the extent of aggregates;
- a subroutine CALL statement including a list of arguments;
- macro calls which key invocation of special routines in the decompiler to handle idiomatic expressions. The arguments to the macro can be used in the translation;
- labels indicating a possible control path join point which might not be reachable except via a computed GOTO operation or from some external routine;
- symbolic constants, indicating the type of variables and constants, which aid the decompiler

in generating declarations;

The output listing of an assembler not only contains the source program itself, but also usually additional information such as machine code, statement sequence numbers, symbol tables, cross-reference information, and macro expansions. This form of the input program seems ideally suited for use by a decompiler, but the volume of the information presents problems that would not be encountered with a more compact input form such as machine language alone.

Formatting the Input

A preprocessor can handle the details of interpreting the raw format of PL be it a core dump, load file, assembler output listing, or some higher level language program. If cross references or symbol tables are to be input to the decompiler they must also be formatted. Such a preprocessor is source language dependent but by translating the input into a standard form it buffers the later portions of the decompiler from input format details.

The exact format of input to the decompiler is not important but the format should be easy to interpret and contain all of the information present in the original form that would be of use to the decompiler. In the next chapter, a particular input format for assembler listing information is described.

LOADING THE PROGRAM

Pragmatically, the entire symbolic text of most programs of interest to the user of a decompiler are too large to fit in the high speed memory of many computers. Thus the decompiler might have to examine subportions of the text. The maximum memory space available to the decompiler defines the size of the "window" through which it can examine the source text. Since in a real sense the window size determines the amount of information a decompiler can "know" at one time about the text in the window, several questions arise:

1. Should the source be passed under the window more than once as is done in multi-pass translators?
2. How is the information flow past the window managed to facilitate natural segmentation of the program text?
3. If analysis requires information outside the window, what is done?

Multi-pass or Not?

We might consider the multi-pass approach to decompilation because in a single pass system there are certain kinds of information which are needed before the text containing that information is read. Examples are symbol definitions, variable usage information, and call/return protocols in subroutines. The multi-pass approach allows the decompiler to collect this information before its starts generating code, but more than one pass through the input data will most probably cause the execution time of the decompiler to be longer than in a one pass system.

The approach taken in the experimental implementation described in the following chapters is to pass through the program once. Symbol definitions are extracted from the symbol table in the listing from the assembler. Information about variable usage that cannot be retrieved is defaulted. Subroutines called from code segments under analysis are considered black boxes which have read/write access to <u>all</u> variables. Information about call/return protocols is requested from the user.

<u>Segmenting the Program</u>

We would like the physical program segments in the window to represent a logical segment that can be treated as a unit. The loading algorithm should be able to recognize, from clues in the source text, the start and end of certain logical units. For example, the most natural logical unit to look for is the subroutine. If the start of a subroutine is indicated by the use of a particular assembler pseudo-op (e.g., ENTRY), then the loading algorithm can cease loading the last unit into the window when it sees such a statement. (This type of heuristic is not applicable to pure machine code without a symbolic source, because such clues do not normally appear in the code.) After the information in the window ~~is~~ processed, the loader proceeds to load the next unit. Of course, given any window size, there are logical units which can exceed that size. The analysis algorithms must handle those cases where the window is too small, but perhaps with decreased ability to recognize control structures and variable usage in code which is not entirely contained within the window.

Information Sources

During the decompilation process, information about the code segment under inspection is available from three sources

1. previously acquired information -- from examination of previous program segments plus information provided by the preprocessor, other passes through the data, or other information sources;
2. the window -- the program segment itself;
3. the user -- the decompiler asks the user to supply the information it requests. (The user is interrogated when the data needed is not available from the other two sources.)

Information needed by the decompiler can be classified as helpful or necessary. The distinguishing characteristic between these two classes is that helpful information is not absolutely required whereas necessary information is required.

Examples of helpful information are cross-reference tables, comments, some kinds of variable usage information, and symbolic names. If helpful information is not provided, then the decompiler has a reserve strategy which can be invoked. This strategy assumes the "worst" response to a request for information. Regardless of the assumption, however, lack of helpful information does not cause an error to be committed. An example of such a case is a request for information about the busy status of a variable on entry to a subroutine. The "worst case" assumption for purposes of substitution procedures might be "busy".

On the other hand, lack of necessary information can lead to errors in some analysis or code generation procedure in the decompiler. When the need for such information arises and it cannot be supplied by an information source, a reserve strategy of assuming a "probable" response and continuing might be acceptable in most instances. (Of course, the decompiler could simply give up, but usually some output is better than none at all, particularly when assumptions made by the decompiler are clearly labelled "caveat emptor".)

An example of necessary information needed by the decompiler is given in the following scenario -- Suppose the program under inspection has subroutine calls to procedures which never return to the calling routine. In effect, calls to these routines are logically similar to simple transfers of control. Normally, control flow after a call to subroutine resumes after the call but a call to a routine which does not return could be followed by a code (or data) segment which is not a logical successor of the call instruction. If information regarding the question of whether a subroutine will return after it is called is not available, then the decompiler might choose to assume that it did return and thus cause the construction of an incorrect control graph.

REPRESENTATION QUESTIONS

The semantics (or meaning) of the statements in the source program must be represented to the decompiler. A natural representation of such information has developed in computer science -- the graph. The nodes of the graph contain information about the functional transformations, evaluations, and movements of data. The edges of the graph represent the sequencing of those operations. A flowchart is a familiar form of such a graph.

Assume that the decompiler uses a graph to represent the source program. We have two questions we need to answer. How is node information represented? How are the interconnections determined?

Node Information

If the meaning of a statement in some language is defined by the interpreter of that language, the choice of node representation is really a choice of an interpreter for that representation. One representation we could use is the input language itself. The interpreter of the statements in that language is (presumably) well-defined corresponding to some translator, executor pair (e.g., an assembler and a machine). This choice binds the decompiler to a particular source language and machine, but we would like to apply a decompiler to a variety of input languages without rewriting the routines which interpret the meaning of node statements.

Another alternative to the representation of the meaning of source statements is to standardize the language in which the node statements are written and then translate all input programs into this standard language. This intermediate language should be simple so that the interpreter of the statements can be simple. If, on the other hand, this standard language is too primitive causing one input statement to be converted to many standard statements, then the efficiency of the decompiler will be impaired. In the following chapter an intermediate language representation is proposed and explained.

Control Information

Control flow is determined by interpreting the instruct~~ions~~ in the program insofar as they would affect the control element of the (source) native machine. Machine defined control sequencing interpretation can be built into the control graph generation routines and tables. This information can be found in the machine reference manual and is machine dependent. Examples of the type of sequencing operations present in most computers are

1. "do the next instruction" (default sequencing),
2. halt,
3. explicit jump (GOTO) conditional or unconditional,
4. jump to subroutine and save return address.

Default Sequencing. Control interpretation is affected by the size of the instruction executed. In a machine with single word and double word instructions the program counter advances to the next instruction by adding the instruction size to the program counter.

Halts. Halt instructions are not common in many programs, particularly ones designed to run in a timesharing environment. These instructions may or may not have a next instruction depending on whether it is meaningful to press the "RUN" button on the machine when a halt occurs.

Multiple Successors. A node which has more than one possible successor has more than one edge directed from it. In the case of simple conditional transfers the two possible successors of the conditional instruction are easily determined from the transfer statement. There are many of these statements in a common program. We explicitly represent the choice of the two alternate paths from a conditional node by labelling two edges directed from the node according to the value of the selection function, e.g., "true" and "false".

For a computed GOTO-like transfer, there may be many possible successors of the transfer node determined by the selection function. The procedure for discovering the selection function as well as its domain and range is non-trivial. The best we can do would be to discover the potential transfer targets by analyzing the possible values which could be taken by the computed transfer index. We label the edges emanating from the conditional node with the value (or predicate) of the selector function which would cause that path to be taken.

Subroutines. A serious problem in simulating the control sequencing of a program is the determination of the control attributes of a subroutine call. This is involved with the subroutine protocol and linkage conventions used by the programmer who created the code. There may be several different protocols used throughout the program. When the decompiler interprets a subroutine call, it tries to retrieve a set of attributes associated with the subroutine invoked and uses this set to determine where the next in-line instruction is. This subroutine attribute list must be created from some knowledge which can be deduced from the program by the decompiler or by a human. If such an attribute list is not available, then the decompiler assumes a default set of attributes and continues.

The subroutine attribute list should contain the following information:

- a list of data items which are used inside the procedure, i.e., they are arguments to some statement in the subroutine.
- a list of data items changed inside the subroutine, i.e., an instruction in the subroutine changes the value of the data item.
- a list of data items which are used <u>before</u> they are changed (if they are changed at all). These are variables which correspond to a notion of parameters or global arguments. The values of these variables are generated outside the subroutine before entry.
- a list of local variables. These are variables which are initialized before they are used in the subroutine and their value is not needed by any

calling routine after the subroutine exits. These correspond to temporary variables of the subroutine.

o a list of variables whose values are needed after the subroutine exits. These correspond to returned values or global variables set by the subroutine for later use.

o how to calculate the "return" address for the subroutine and where to put the value. In general, the return address is information which tells the subroutine where it was called from and is related to the next item,

o the return point from the subroutine or how to compute it given the return address and other information. A normal return from a subroutine would be to the return address given it by the calling procedure, usually the address just after the call. In some cases, however, the return address may be computed at run time by the subroutine. A common example of this is the so-called "skip return". The subroutine call is followed by an error exit instruction which is skipped over by the return if an error did not occur. A return past an in-line argument list (of possibly variable length) is another example of a computed return address mechanism. The subroutine may not return at all, in which case a subroutine call is more like an unconditional jump.

The information about the variable usage is needed when a subroutine call is encountered while analyzing the control graph for condensation of subexpressions. The information about the return address is needed to determine where control flow begins again after a subroutine call.

Graph Implementation

The implementation of the representation of the control graph greatly influences the ease and efficiency with which control graph transformations and analysis may be carried out. For example,

- o the process of finding the successor(s) of a node in the graph should be a simple operation;
- o two-way links should be considered for representation of the edges so that access to the predecessor(s) of a node is also easy;
- o the information in the node itself should be easily read and modifiable;
- o the inclusion of new structures and data generated by the decompiler should be allowed.

PROGRAM TRANSFORMATIONS

The creation of the graph representation of the program involves considerable effort interpreting the control structure of PL and translation of PL statements into a form to be used in the nodes of the graph. Once this representation is built, there are natural transformations of the graph which are useful in the translation to a large number of target languages.

The application of transformations to PL must preserve the functional meaning of the program. Given the same inputs, PL and PH should compute the same outputs. (This condition might be relaxed if the output of the decompiler was not a program to be executed.) What does not have to be preserved is the <u>exact</u> method of computing the output values. The possible transformations on PL could range from simple transliterations (identities) to the complete substitution of one algorithm for another (.e.g., a binary sort substituted for a bubble sort). I do not propose that a decompiler can or should operate at either of these extremes.

One of the distinguishing characteristics between low level languages and higher level ones is their differing ability to represent the procedure for performing a logical unit of computation. In low level languages expression evaluation usually consists of a linear stream of primitive instructions specifying in detail the data movement and functional transformations needed to compute the desired value. The problem with such a description is that the text contains information about <u>how</u> the expression is to be calculated relative to the rules of a <u>primitive</u> interpreter.

Higher level language statements are instructions to a more sophisticated interpreter. Less information about how to perform the details of a computation needs to be explicitly stated in a higher level language. These primitive operational details should be suppressed in the output of a decompiler.

We list some of the more general implementation details which commonly appear in low level language programs

- use of temporary variables,
- concerns over data storage in fast registers versus slower main memory,
- conditional or iterative control decisions based upon peculiarities of a particular instructions set,
- movement of data due to incomplete data paths between memory elements (e.g., lack of memory-to-memory operations causes movement through registers).

We distinguish items on the above list from others such as I/O programming, word length considerations, different internal representations for data items, etc. The former characteristics of a program are not formidable bars to decompilation whereas the latter are. (See the following discussion of the problems of transferability.) We refer to the former list as soft implementation characteristics, and the latter as hard implementation characteristics. A successful decompiler deals with these soft problems in a standardized (non ad hoc) manner.

We list some of the transformations upon PL (its graphical form) which deal with soft implementation details:

1. condensation of separate subexpression evaluation code segments into a more nested representation;
2. translation of series of conditional statements into a logical expression;
3. standardization of control sequences for iteration;
4. "repackaging" of bulky code segments as subroutines;
5. node splitting and synthetic substitution to avoid undesirable transfers of control;
6. translation of idomatic expressions into a standard form.

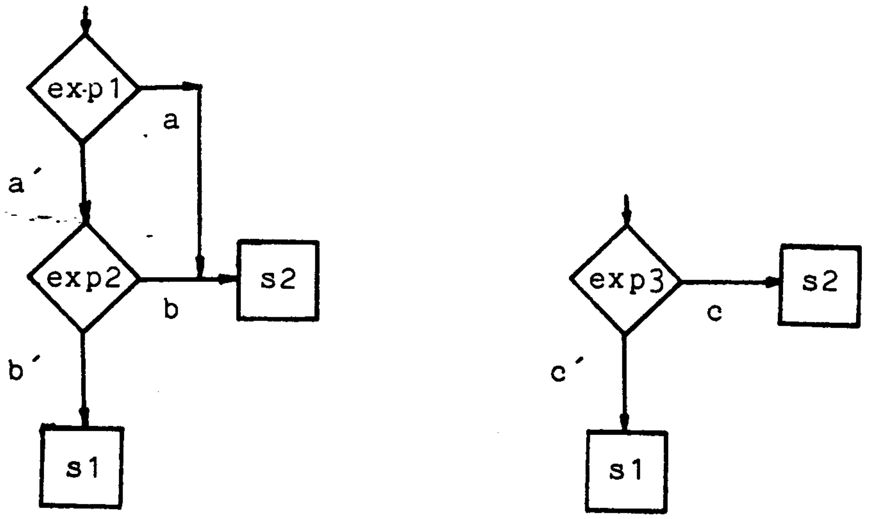
Expression Condensation

One of the more important tasks of a decompiler is to combine data movement and operation statements of the low level language into expressions in the higher level language. Figure 1-A shows an example of this kind of condensation. Chapter 4 discusses the algorithms necessary to perform this condensation. These algorithms are based on analysis of the usage of variables in the low level program. If the temporary uses of variables can be identified, they can be eliminated. Most temporary uses are naturally in the registers of the source program. However, if a general approach is taken to the analysis of data movement so that main memory and fast register distinctions are eliminated, temporary uses of non-register variables can also be detected. This is particulary important on machines with a small number of accumulators where the usage of main memory to hold partial results is common.

Logical Expressions

Conditional flow of control through a low level program is often the concrete manifestation of a computation involving logical (Boolean) values. The task of converting control statements into logical expressions in a higher level language is more difficult than the task of expression condensation. It is a superset of that problem. The execution independence (defined in chapter 4) of control statements and data manipulating instructions must be determined if we expect to condense any but the most trivial logical expressions.

The basic logical expression condensation techniques involve recognizing some simple control patterns in the source program. Figure 3-A illustrates a pattern of control based on serial tests which can be combined into one test with a more complex conditional expression. Various translations selected depend upon the sense of the tests involved and the desired sense of the resultant test.



a	b	exp3 (c=true)	exp3 (c=false)
false	false	exp1´ or exp2´	exp1 and exp2
false	true	exp1´ or exp2	exp1 and exp2
true	false	exp1 or exp2´	exp1´ and exp2
true	true	exp1 or exp2	exp1´ and exp2´

Figure 3-A. Combining serial tests. Formation of resultant values given sense of the tests.

Of course, the transformation of Figure 3-A can be applied recursively to condense a series of more than two conditional tests into a single test.

Standardization of Iterative Control

One of the difficulties in understanding the iterative flow in low level languages is the fact that the programmer usually has an almost unlimited selection of techniques and instructions at his disposal when he creates a loop. The higher level language may have only one type of iteration statement. If this is the case, then a decompiler should be equipped to perform the translation of a set of different iterative control structures into the standard sequence in the target language.

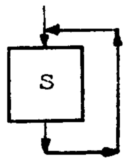
For example, suppose that our target language has only one type of iterative statement -- one with a pre-test. Let this statement be a WHILE statement of the form:

WHILE exp DO statement.

Consider source program loops of four kinds -- no test for completion (infinite loop), test at top of loop (pre-test), test in the middle of the loop (mid-test), and test at the end of the loop (post-test). (The infinite loop appears in systems programs where the occurrence of an external event such as an interrupt causes an exit from the loop.) Figure 3-B illustrates flowcharts for each of these four loops and their translation into the target language.

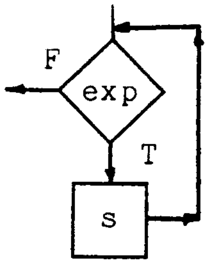
In the case of the infinite loop, a variable with the constant value of true is inserted in the conditional expression test of the loop. The pre-test loop translates directly into the target language. The loop with the test in the "middle" can be translated by a process of node-splitting (S1 is duplicated). A similar process takes place when the test is at the end of the loop.

infinite loop



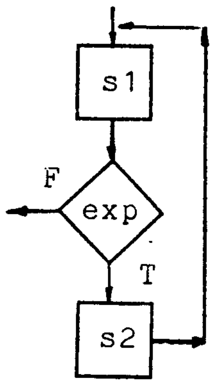
```
WHILE true DO s;
```

pre-test loop



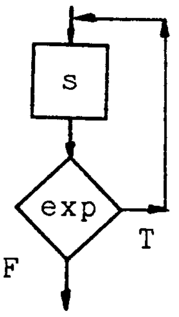
```
WHILE exp DO s;
```

mid-test loop



```
s1;
WHILE exp DO
 BEGIN s2; s1 END;
```

post-test loop



```
s; WHILE exp DO s;
```

Figure 3-B. Looping control structures.

The process of node-splitting or copying nodes in the body of the target program can be achieved two ways -- the code can be physically copied so that it appears in two locations, or it can be put into a subroutine and calls to this synthetic subroutine are generated in the appropriate places. The choice of which approach to use depends heavily upon the volume of code in the copied statements. A large volume of code should probably not be copied, but rather put in a subroutine. On the other hand, if the statement were a primitive in the target language, such as a simple assignment statement, then the duplication of the code could be tolerated. (Node splitting is a common technique used in the optimization phase of a compiler. Aho and Ullman (1973) describe this technique.)

Repackaging Code Segments

For readability, the size of the code controlled by a conditional test or iterative test should not exceed a "comfortable" maximum. This maximum is a subjective quantity, but I suggest from experience that the body of an IF statement, for example, should probably not exceed thirty or forty lines or about one-half of a line-printer page. If the body gets much larger than that, the physical position of the controlling expression is not "close" to much of the code controlled.

In the University we try to encourage students to write modular programs using subroutines and functions in order to reduce the size of any one portion of code. The same philosophy can be programmed into a decompiler so that when a single entry, single exit sequence of code exceeds a certain size one or more portions of it are placed in synthetic procedures. This approach becomes of particular importance when the nesting level of certain expressions such as IF...THEN...ELSE becomes greater than two or three and the code contains tens (if not hundreds of source lines as each level.

The suggestion is to try to repackage the code segments into smaller portions, when the original programmer did not use a modular programming technique. The critical question is where should the code be broken? Can any measure be used to determine a modularization process that will be any better than a random choice? This subject relates to the problem of partitioning digital circuits into sub-components on several printed circuit cards. This hardware design problem has been reviewed by Breuer (1972). None of the decompilation systems reviewed in the course of preparing this dissertation (see Chapter 2) have attempted the solution to this problem. The experimental decompiler described in later chapters does not perform this task either. The topic has been left for future research.

GOTO-less Target Programs

In the discussion of standardization of iterative control we mentioned the process of node splitting used to transform an iterative control structure of the source program into a standard form. Another process of standardization, the elimination of GOTO statements from the target program, deserves some mention.

Knuth (1974) elegantly summarizes the debate which has raged the last few years over the GOTO statement in programming languages. He lists 102 references to the literature on this subject for the reader who wishes to investigate the matter in great detail.

In our discussion of this matter, let us assume that we wish to avoid generating <u>any</u> occurrences of GOTO statements in the target program output by the decompiler. The node splitting techniques help us when we have loops involved but node splitting by itself cannot help us standardize certain control graphs. (See Aho and Ullman 1973). The introduction of synthetic control variables may be necessary. The state of the control variable is altered to cause certain paths through the program to be taken when it is tested in a conditional target statement.

An example of the introduction of a synthetic control variable to remove a GOTO statement is given in Chapter 4. The real question is whether a decompiler should seek to achieve a goal of creating GOTO-less target programs. The weak structure of low level languages almost assures that the average programmer will write many control structures which would need to be handled by such a mechanism. In the node-splitting and synthetic control variable processes, code would be created in the target program that had not appeared in the source. If the decompiled program is used for production purposes, one must consider the extra run-time computation introduced into the target program. In any case, as Knuth (1974) points out, we can expect the result of removing the GOTO statements from a badly structured program to yield a badly structured program.

Idiomatic Expressions

Housel (1973) cites a definition from Gaines (1965) which describes an idiomatic expression as "a sequence of instructions which form a logical entity, and which cannot be derived by considering the primary meaning of the instruction." An interesting feature of a decompiler is the recognition of these idioms in the source program and the translation of the meaning of the statements rather than a literal paraphrase in the higher level program.

If the source and target machines are very similar, it is reasonable to expect that they will have many idioms in common. In that case, a literal paraphrase would not be incorrect in the target program, but would be less desirable than a more general translation of the idiom. An example of this type of idiom occurs on machines with two's complement arithmetic. The two's complement value of a register can be calculated by taking the one's complement of the register and adding one. Some machines provide this function as a primitive operation, some do not. Another example is the complement instruction itself. Some instruction sets do not have a complement instruction, but rely upon the exclusive-OR of a word of all ones to perform this operation.

An experienced assembler language programmer will have a repertoire of many idioms in his bag of programming tricks. Some of these may be codified in macros; most are usually not. The more general and often used simple idioms (such as those mentioned above) can be recognized by the decompiler and converted into a standard phrase in the target language.

CREATING THE OUTPUT PROGRAM

The decompiler should translate patterns of PL instructions into the highest level constructs for PH commensurate with program understandabilty. The kinds of control structures to be generated are, of course, related to the high level target language but some typical patterns to be recognized and dealt with are expression evaluation and assignment, conditional statements, conditional and iterative loops, subroutine calls (including argument passing) and returns.

The main reason for choosing an intermediate language representation for the input program is to provide some independence of the internal decompiler processes from the details of the particular input language. The analysis routines manipulate the intermediate graph structure. The output routines translate this graph structure to the target language. We have the problem of translating the control structure information and node information while providing some independence of the intermediate form from the target language.

The use of table driven procedures and localization of information regarding the specific target language can yield output routines which are easily adapted to a wide range of target languages. In the following chapter we discuss the design of such a set of output routines.

PROBLEMS OF TRANSFERABILITY

The translation of the recognized constructs, both data and control, into the higher level language requires the mapping of these patterns to an equivalent construct in PH. In the case of translation for understandability, the equivalent structures can be whatever representation is the most convenient for illustrating the behavior of PL. Since a human being is the interpreter of this output, details of implementation can often be suppressed in order to present the more general aspects of the algorithm.

In the case of decompilation for transferability, the problem is more difficult, especially with respect to data structures. Two concepts to be mapped from PL to PH are positions of data items in relation to others in memory and the length of the data fields. A mechanical interpreter must be presented with a procedure which it can follow to generate the appropriate outputs given inputs in the domain of PL. Each small side effect which may influence the output of the procedure must be faithfully reproduced. This is the general reprogramming problem discussed in Chapter 2. These problems influence the quality of the output of the decompiler along several dimensions

- o time and space -- often less efficient algorithms must be used on the target machine;
- o readability -- contortions appearing in PH to simulate some feature used in PL may obscure the purpose of the code;
- o unrepresentable algorithms -- some statements appearing in PL may not have any analog in PH (e.g., I/O instructions).

Because of the difficulties of achieving a general translation which is acceptable as a production program, the use of a decompiler for transferability can only lead to success if the interpreters for PL and PH are "well-matched". This means that the differences between the interpreters in the area of hard program implementation characteristics are minimal. It is not sufficient for the higher ~~level~~ interpreter to cover the lower level one. For a production program it must cover the low level implementation features efficiently.

Word Lengths

Differing word lengths pose a particularly difficult problem for the decompiler. Either the word size of the source machine must be simulated on the target machine (time-consuming) or the algorithms and data structures must be translated into something equivalent. For example, a simple counter in PL whose value does not exceed 100 can be easily represented in practically every computer ever built. If the range of values in the counter was 0-100,000 then the 16-bit word of a mini-computer would not be capable of storing the value.

Assume we have a program PL which was written for a machine with a native byte length of six bits and the target machine on which the high level program is to be translated has an 8-bit byte. What do you do? Do you assign each 6-bit byte to occupy one 8-bit byte and let the other two bits per byte go unused? This would seem to be a natural solution to the problem. However, what if the low level program used the fact that the rightmost bit of the nth byte was physically next to the leftmost bit of the n+1st byte?

Are we reduced to simulating the 6-bit/byte feature on the 8-bit/byte machine by creating subroutines and artificial data structures which do not fit in with the native mode of the target machine? That is what is done when we choose a high level target language which has facilities for the description of arbitrary data structures which do not match the primitive structures and accessing methods of the hardware. The attendant inefficiency associated with the execution of the PH representation of PL program would make decompilation for the sake of transferability of most production programs an unattractive alternative to emulation. (Note that emulation is interpretation at the hardware or micro-code level.)

Other Machine Dependencies

Most programs use only a small number of tricks or features of the interpreter which make them truly machine dependent algorithms. A human being who wishes to translate the program to another machine of dissimilar characteristics searches the program for these and recodes the algorithm in another manner which is amenable to the architecture of the target machine. For example, consider the translation of a program written for a 16-bit minicomputer which has byte addressing, to a machine similar in other respects except that it has a word addressing mode only. A human would examine the program to be translated for all of the expression calculations which are address computations and adjust them to reflect the word addressing mode of the target computer. He might change loop increments from two to one for a word array access. All byte accessing instructions might be changed to simple subroutine call or in-line code consisting of load, and shift or mask operations. The representation of byte pointers might have to be changed to reflect the word addressing architecture, but the bulk of the program logic is unaffected by the macnine differences.

If a decompiler can be programmed to simply recognize the hard implementation characteristics of PL, and to fix the ones it can and flag (or announce) the ones it cannot, the job of transferring the program would be much easier for humans. Of course, the more similar the architectures of the two machines, the easier the task. As they get more dissimilar the task becomes more difficult until manual reprogramming, emulation, or simulation is the only recourse left.

STORAGE STRUCTURE RECOGNITION

The kinds of data access patterns which should be recognized are indexing, indirection, bit, byte and word accesses, as well as data structure accesses to stacks, queues, deques, lists and arrays. Some data structures may be equivalenced in PL, i.e., one structure may partially or totally overlay another. For example, an array of items may be treated as a string of bits one time, floating point variables another time, and bytes of character strings in yet another part of the program. Obviously, the kind of manipulations of data items allowed by low level languages are often difficult for a human being to properly interpret so we should expect a decompiler to have difficulties in this area also.

Patterns of data access are only a clue to the extent of a data structure such as an array, for example. Given no other information, a set of simulations or the symbolic execution of the program PL could indicate the total possible set of values which an index variable might assume during a computation. A static determination could be made in some cases (for example, a loop with constant initial and final index values). The symbolic declarations of an assembler language program would be helpful in determining the extent of a storage structure.

STATISTICS GATHERING

The decompiler, in addition to creating the target program, can gather statistical information about PL. Such information might consist of the number of loops, jumps, special idiomatic patterns recognized, frequency of untranslatable code elements, frequency of reference to certain variables, scope of variables, memory maps indicating code and data areas, types of argument passing to subroutines, etc. Since the decompiler is handling all of this information and more in the normal course of the translation, it should be simple to instrument the decompiler to accumulate any information it may have into a summary form to be displayed for the user. Such information should reflect in some quantitative way the difficulty the decompiler had in translating PL. This information can be used to lead to improvements in the decompiler or as an evaluation of the programmer who wrote PL.

SUMMARY

In this chapter I have tried to present a synopsis of the many design considerations associated with the construction of a decompilation system. This chapter was a natural outgrowth of the design work which went into the experimental decompiler described in Chapter 4. It has been an attempt to provide a checklist of items which should be reviewed before a decompilation effort proceeds to the implementation stage.

Chapter 4

AN EXPERIMENTAL DESIGN

INTRODUCTION

This chapter discusses the design of various parts of an experimental decompiler built during this research. The decompilation process can be considered to consist of seven steps:

1. preparation and formatting of the source program;
2. loading of the source program into the decompiler;
3. creation of a directed graph representing the control flow of the program;
4. translation of the source program instructions into an intermediate "machine independent" code;
5. analysis of variable usage for expression condensation;
6. control structure recognition and translation to the target language;
7. postprocessing of the output of the decompiler to accomplish any automatic editing or manual changes.

We will treat each step in turn, focusing on the general algorithms and data structures developed without subjecting the reader to language or machine dependent details.

Algorithms described in this dissertation will be written in a notation which is a natural mixture of several modes of presentation. The purpose of this notation is to convey to the reader the basic mechanisms of the process involved in the algorithm without burdening him with implementation details. Following chapters will discuss the particulars of the implementation along with the results of some experiments on real production programs.

PREPARATION OF THE SOURCE PROGRAM

As discussed in Chapter 3, the preferred input to a decompiler is the symbolic source program so that symbolic names and comments may be preserved by the decompiler. The output listing of an assembler is well suited to this use. To buffer the later portions of the decompiler from vagaries in listing format, a front-end routine is needed to arrange the data in the form expected by the decompiler.

Consider a prototype format for the listing output of an assembler. Each statement becomes a list of the following elements:

```
<sequence number>
<location address>
<label>
<opcode>
<operands list>
<comments>
```

<value list>

where

<sequence number> is an index indicating the relative physical position of the assembler statement with respect to the rest of the statements in the program;

<location address> contains the address where code generated by the statement is located. If no code is generated, this field is empty;

<label> contains the symbolic label(s) used on this statement. If no label is present, this field is empty;

<opcode> contains the symbolic opcode of the assembler language statement;

<operands list> is the list of symbolic operands associated with the source statement;

<comments> is the on-line comment string, if any; and

<value list> is a list of the machine code generated by the source statement (sequentially starting at <location address>).

For example, assume the source statement

```
L43  STAE  BLOC,1   SAVE LOCATION IN TABLE
```

when assembled in context with the rest of a program is source line 110, generates code at location 34725:

```
006055
035310.
```

The preprocessor would generate a list of the form

```
'(110,34725,L43,STAE,(BLOC,1),
 "SAVE LOCATION IN TABLE",
 (006055,35310)).
```

A simple comment statement on line 400

```
*THIS IS A COMMENT
```

would generate a list of the form

```
(400,NIL,NIL,NIL,NIL,"*THIS IS A COMMENT",NIL).
```

Using the assembler source listing output, each line is put into the prototype format by the preprocessor. (A similar approach can be taken to format raw machine code. The prototype format would not have to be so complex. It could consist of (address, contents) pairs. This approach was taken early in this decompilation effort and worked well.)

In addition to the listing of the source program, many assemblers provide useful information in the form of symbol tables and cross-reference tables. If desired, these can be formatted and provided as an additional source of input to the decompiler.

LOADING THE DATA

Once the source input has been formatted, the information can be presented to the analysis portions of the decompiler. As each formatted list is read into the decompiler, it is put into two vectors, the memory vector

(M) and the source vector (S). The memory vector acts as a pseudomemory -- it contains the code present in the value list of the input, i.e., the machine language and data generated by the assembler. With each memory value is a pointer to the source vector element containing the source code which generated the memory value. The source vector contains the symbolic label, opcode, operands list, and comments for each statement. We define M and S more precisely,

$$M = \{ m[i] = (\text{value}, \text{Spointer}) \mid i = 0, \text{Mmax} \}$$

$$S = \{ s[i] = (\text{label}, \text{opcode}, \text{operands list}, \text{comment}) \mid i = 0, \text{Smax} \}.$$

Mmax and Smax indicate the size of the vectors. Two other variables, Mreloc and Sreloc, provide for the mapping of the index of the real memory locations and source lines into M and S as is shown in algorithm LOAD. These variables serve as relocation factors. The vectors M and S comprise the window on the source program. The decompiler examines the source program through this window.

Algorithm LOAD. Given an input line L from the preprocessor, we load the information of L into the vectors M and S. (For clarity, we assume M and S are not full.)

```
1. Compute the index on S where we put the source
   code.
   p := <sequence number> - Sreloc.
```

2. We move the data from L to S.

   s[p] := (<label>, <opcode>, <operands list>, <comments>).

3. If the <location address> is empty, then return. (This source statement generated no code.)

4. Compute the index M where we start storing program code and the pointers to S.

   q := <location address> - Mreloc.

5. We store the code and source pointer pairs in M.

   m[q+k] := (kth item on <value list>, p) for k=0, 1, ... ,(number of elements on <value list>) -1.

6. Return.

The problems of window size discussed in Chapter 3 are addressed by the loader by looking for subroutines headers and page eject pseudo opcodes to find natural breaks in the logical-physical structure of the program. The window (M and S vectors) is loaded until one of these breaks is hit or until the window is full. The program segment in the window is then processed and the window is reloaded again. Figure 4-A shows the contents of M and S after loading some sample input.

```
(110, 34725, L43, STAE, (BLOC, 1), "SAVE LOCATION IN TABLE",
                    (6055, 35310))

(111, 34727, NIL, TZA, NIL, "CLEAR AREG", (5001))
```

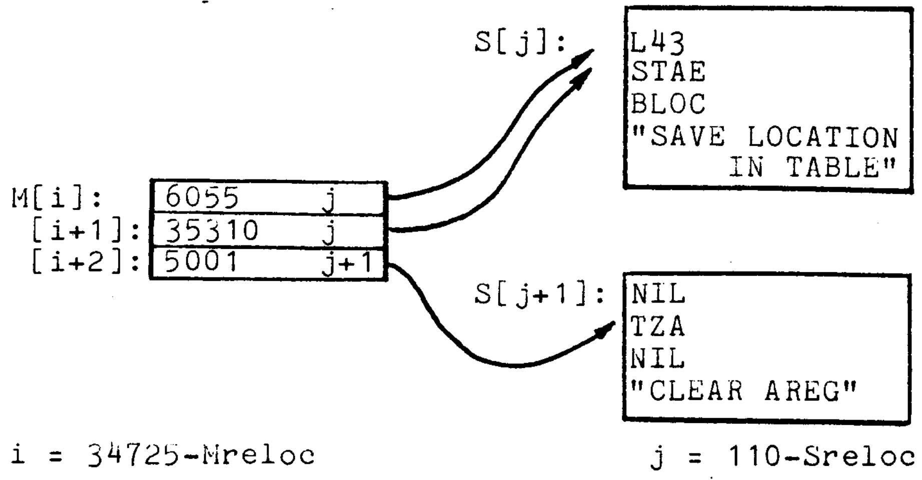


Figure 4-A. Memory variable (M) and source variable (S) after being loaded with information from source lines 110..111, memory locations 34725..34727.

CREATING THE CONTROL GRAPH

A decompiler must be able to determine the flow of control through the source program for two reasons:

1. to determine how the executed instructions affect the data of the program, and
2. to recognize control patterns in a weakly structured program.

The control flow in a program is implicitly determined by the source machine which executes a program written for it. One method of making this information explicit is to create a control graph where the flow to the next instruction is indicated explicitly by an arc in a graph and the instructions themselves correspond to the nodes. Flowcharts are familiar forms of such control graphs.

Stage One Control Graph

The control flow graph created by the decompiler can be considered to be, in the first stage, a directed binary graph, where each node in the graph corresponds to an instruction of the source program that has been "visited" by the graph creation algorithm. We assume for the following discussion, that these nodes can be classified into four types:

1. normal node, one logical successor -- this would represent all instructions which have one and only one successor in the control sequence (unconditional transfers excepted);
2. transfer node, one successor -- corresponds to an unconditional simple jump.
3. conditional node, two possible successors -- this would represent a conditional branch or skip instruction;
4. sink node, no top-level successors -- this would represent an instruction which halts the machine, performs indeterminant jumps (e.g., whose target is computed at run time as illustrated by an indexed or indirect jump).

If a node in the control graph has no predecessor, it is called the root node. There is only one root node in a control graph. If a node has two or more immediate predecessors, it is called a <u>join node</u>. A <u>sink subgraph</u> is a subgraph where every leaf of the subgraph is a sink node. (The common terms of graph theory will be used throughout this discussion.)

The program example of Figure 1-A can be transformed into a corresponding stage one control graph of the form shown in Figure 4-B.

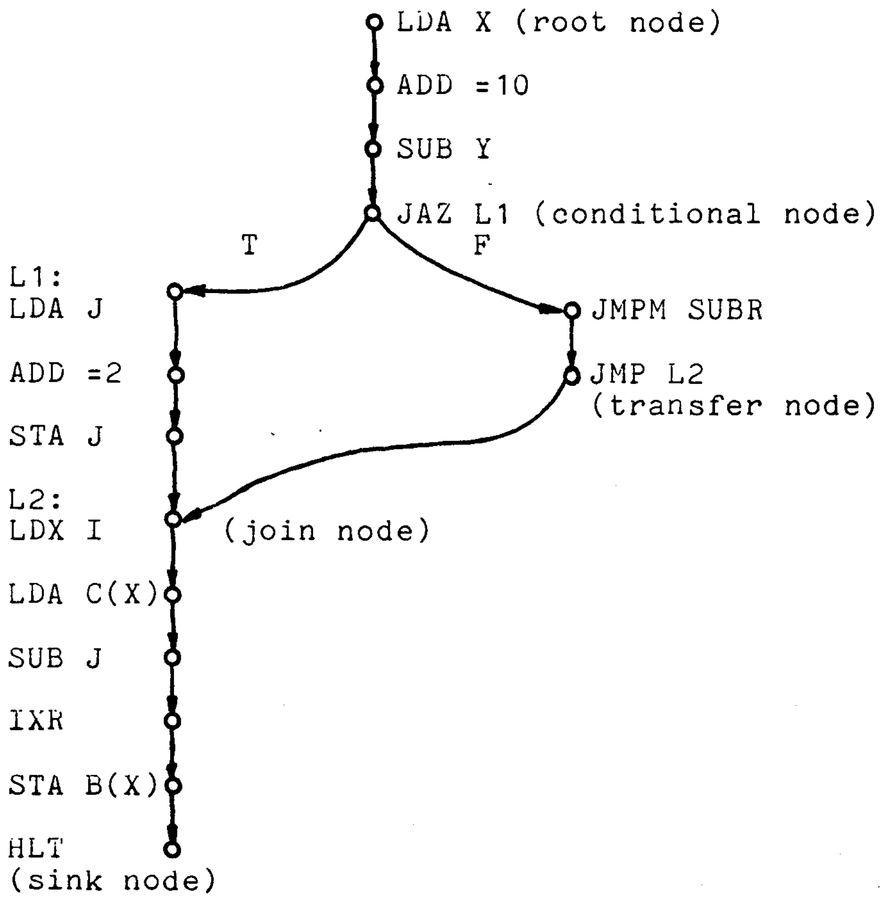
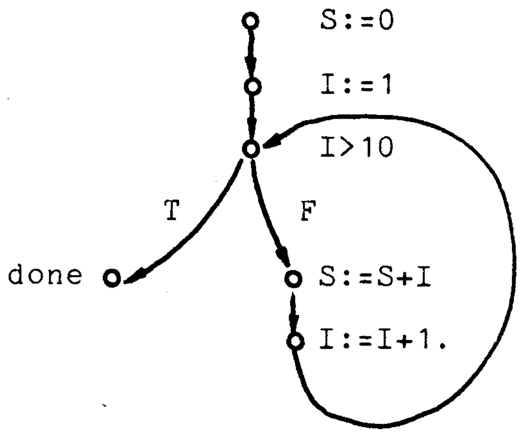


Figure 4-B. Stage one control graph for program segment of Figure 1-A.

The node corresponding to "JAZ L1" is a conditional node. The two directed arcs emanating from it are labelled "true" and "false" to indicate the condition value needed to traverse the arc to the appropriate subgraph. The node corresponding to "L2: LDX I" is a join node because it has two predecessors.

A typical stage one graph structure with a loop link is illustrated for a program to compute the sum of integers from 1 to 10



The STAGE1 algorithm builds control graphs of the kind shown in Figure 4-B and the above example.

Algorithm STAGE1. Builds a stage one control graph. Has one argument, a node x. Returns a control graph rooted at that node.

1. If x is nil, then return nil.
2. If x has been included in a previous invocation of STAGE1, then return a link (join link or loop link) to where x appears in the partial graph.

3. If x is a sink node, then return x.

4. If x is a conditional node, then return the subgraph

$$\begin{tikzcd} & \circled{x} \arrow[dl, bend right] \arrow[dr, bend left] & \\ \boxed{\begin{array}{l}\text{stage1 of "true"}\\ \text{successor of x}\end{array}} & & \boxed{\begin{array}{l}\text{stage1 of "false"}\\ \text{successor of x}\end{array}} \end{tikzcd}$$

The STAGE1 algorithm is invoked for the "true" successor first.

5. x must be a normal node or a transfer node. Return

$$\begin{tikzcd} \circled{x} \arrow[d] \\ \boxed{\text{STAGE1 of successor of x.}} \end{tikzcd}$$

Algorithm STAGE1 obviously needs memory in order to perform step 2 which keeps it from cycling in an interpretive loop. The supporting predicate evaluations needed to determine the type of node as well as the successor function are dependent upon the interpreter of the source language, e.g., the machine hardware, in the case of a real machine language. This information is source machine dependent and must be built into the decompiler in the form of tables or functions in much the same manner as for a software simulator. The successor function may have an undefined value if the successor node is outside the window of the decompiler. In this case, a synthetic transfer node to the remote successor is returned in the graph. This might turn into a GOTO statement in a higher level representation.

The definition of a sink node includes source instructions whose successor cannot be determined by a static trace algorithm such as STAGE1. A program control graph would stop with an indexed jump instruction used, for example, to implement a computed GOTO through a branch table. This essentially cuts off part of the program from the control graph and only program structure contained in the control graph can be decompiled. We are left with a dilemma of serious proportions since indeterminant control transfer is a basic capability of almost every source machine language. What can be done?

Upon recognition of an indeterminant jump, the location of this transfer node in the pseudo memory is remembered for later processing and the control graph creation continues as for a sink node. The control graph processing continues to completion. Then the decompiler is left with tying together the code segments which can be accessed through the branch table.

The approach taken depends heavily upon whether or not the source program to be decompiled is accompanied by symbolic information such as a program listing. If it is, then the decompiler can simply process in a sequential manner the data and code items it cannot reach through the control graph. The first code item it finds which it has not yet decompiled becomes the root of a new control graph. This assumption is, of course, heuristic. However, all of the code segments which were in the original program will be represented in the same sequence, relative to their disjoint control graphs, as they appeared in the original program itself. This strategy assumes that the decompiler can find the first code item not yet included in any previous control graph. This can be done, (again heuristically) by examining the symbolic operation code field of the source instructions. Instructions represented in the source as data items will appear as such in the decompiler's output. The assumption is that if a programmer meant

ADD X

he would have stated it that way instead of as

DATA 0120000+X.

If the source program is more akin to a memory dump, with no symbolic information available, either the decompiler or a human must determine the transfer function of an indeterminant jump and its possible domain and range in order to find the target.

Stage Two Control Graph

It is important in the process of decompilation to identify paths between instructions and, in particular, to identify join nodes. Join nodes represent a convergence of several control paths which may conform to higher level control structures such as loops, or represent the continuation of a common path after following separate branches from a conditional node (e.g., the code sequence following an IF...THEN...ELSE construct).

A stage two control graph is developed from a stage one control graph by analyzing the occurrences of join nodes and rearranging the structure of the graph so that later inspection of the graph by structure recognition routines will be facilitated. The process transforms the binary graph into a ternary one where the conditional nodes and their subgraphs are rearranged so that the portion of the subgraph at and below the join node is attached to the conditional node common to both entering paths. For example, the first stage control graph of Figure 4-B is transformed into that of Figure 4-C.

The stage two algorithm creates synthetic transfer nodes to the join subgraph if such transfer nodes did not exist already and are needed. In Figure 4-C, an explicit "JMP L2" transfer node has been placed at the end of the "true" subgraph. The subgraph rooted at join node "L2: LDX I" is reattached as a third subgraph of the conditional node "JAZ L1." This rearrangement can be interpreted as follows: execution proceeds from the root node to the conditional node.

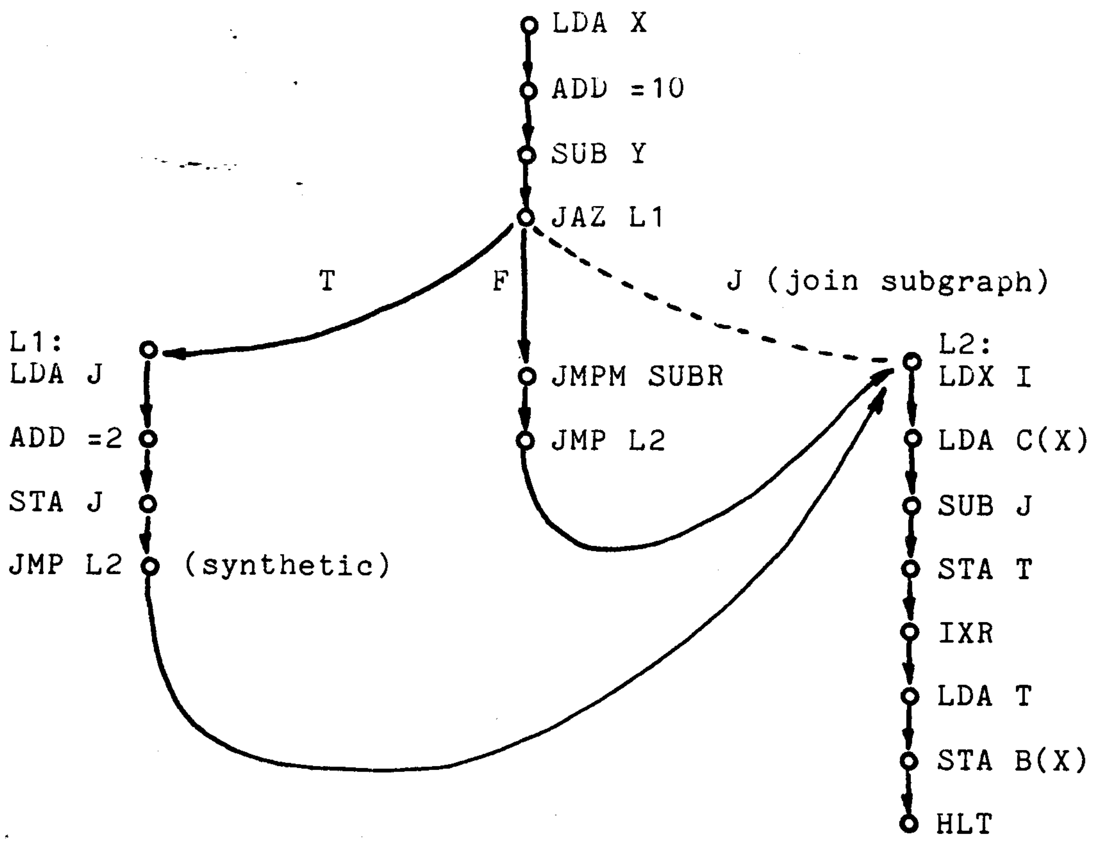
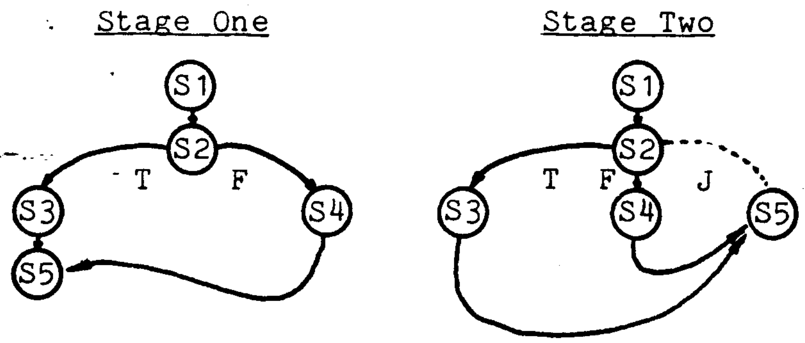


Figure 4-C. Stage two control graph derived from the stage one control graph of Figure 4-B.

If the condition is true, the instructions of the "true" subgraph are executed; if the condition is false, the instructions in the "false" subgraph are executed. In either case, execution of a leaf node which is not a sink node in the "true" or "false" subgraph is followed by execution of the join subgraph of the conditional node.

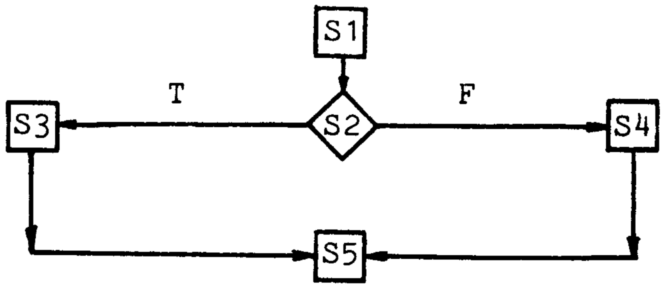
Thus a possible higher level representation corresponding to a stage one and stage two graph of the forms



might be

```
S1;
if S2 then S3 else S4;
S5;
```

The graph above is really a convenient representation of the familiar flowchart form

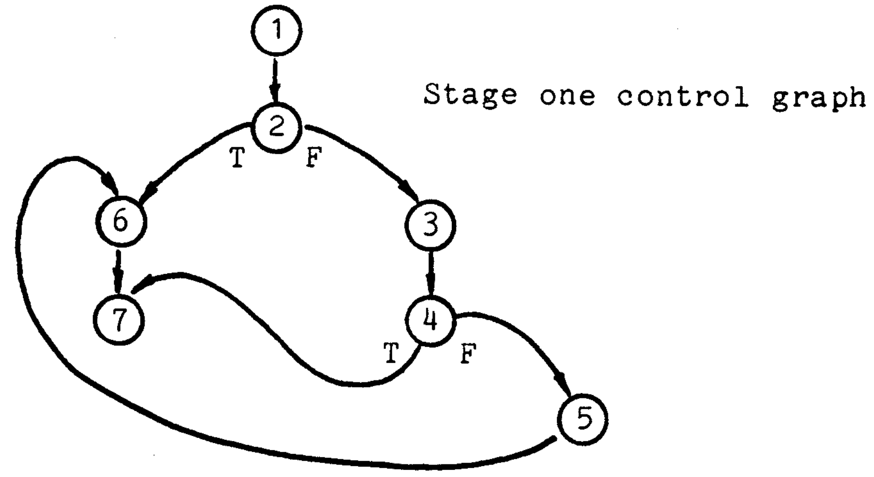


The stage two control graph simply makes explicit the fact that there is a join node reuniting the alternate paths emanating from the conditional node.

It is important to note that in a stage one graph, join links (links from the "true" or "false" subgraphs into the join subgraph) always point from the "false" subgraph to the "true" subgraph of a common conditional node ancestor. This fact is imposed upon the structure by the arbitrary convention to always trace the "true" branch of a conditional node first (see algorithm STAGE1). (A decision to trace the "false" branch first would yield join links directed from the "true" subgraph to its "false" sibling.) The fact that, under this convention, join links may only point from the "false" subgraph to the "true" subgraph of a conditional node simplifies the nature of the stage two graph generation algorithm. The asymmetry of the graph means that certain portions of the algorithm are only required to examine <u>one</u> of the subgraphs of a conditional node to find join nodes linked to the sibling subgraph.

It is sometimes the case, as shown in the example of Figure 4-D, more than one join link from the "false" subgraph to its "true" sibling, e.g., 4-7 and 5-6. Application of the STAGE2 algorithm causes the subgraph rooted at node 6 to become a join subgraph. After application of the STAGE2 algorithm, join links not directed to the root of the join subgraph of their own subgraph root node will correspond to an explicit GOTO statement in a higher level language. In the example of Figure 4-D, the 4-7 link requires a GOTO statement. If a GOTO-like statement is not available, then the graph must be restructured by node splitting (e.g., duplicate node 7 and attach it to node 4), or by the introduction of synthetic nodes which can test or set new predicates and thus force execution of the proper statements.

Code	Meaning
1 LDA X	A:=X
2 JAZ 6	IF A=0 THEN GOTO 6;
3 LDA Y	A:=Y;
4 JAZ 7	IF A=0 THEN GOTO 7;
5 LDA Z	A:=Z;
6 SUB W	A:=A-W;
7 ADD =3	A:=A+3;



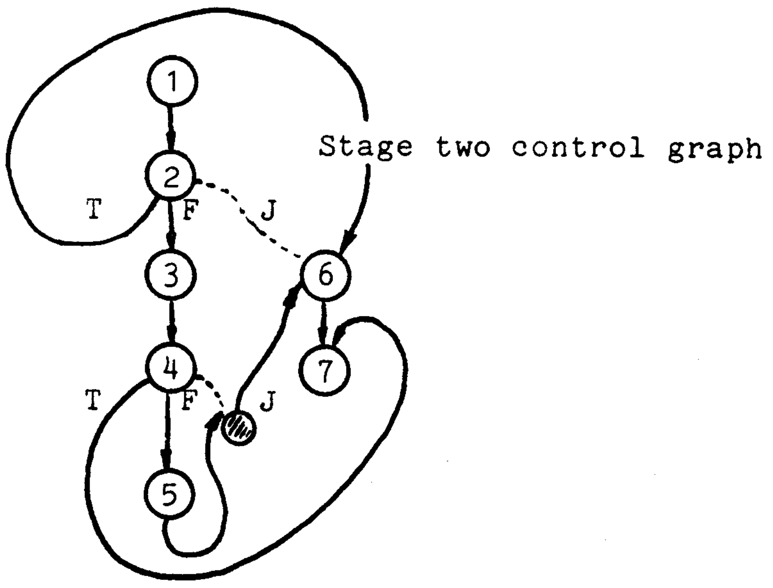


Figure 4-D. Example of links to two different nodes in a join subgraph (2T-6, 4J-6, and 4T-7).

In some higher level language , the GOTO code for the program segment of Figure 4-D might be

```
        IF (A:=X)#0 THEN
           IF (A:=Y)=0 THEN GOTO L7 ELSE A:=Z;
        A:=A-W;
    L7: A:=A+3;
```

With the introduction of a synthetic variable P and an additional test, the GOTOless form of the program would look like

```
        P:=FALSE;
        IF (A:=X)#0 THEN
                     IF (A:=Y)=0 THEN P:=TRUE
                                    ELSE A:=Z;
        IF NOT P THEN A:=A-W;
        A:=A+3;
```

Another possible higher level language representation might be

```
        IF (A:=X)#0 THEN
               IF (A:=Y)#0 THEN BEGIN A:=Z; GOTO L6 END;
               ELSE NULL
        ELSE L6: A:=A-W;
        A:=A+3;
```

This form is unacceptable because of the branch into the scope of a conditional statement. The stage two graph generation algorithm (STAGE2) prevents the creation of such a structure so that there are no join nodes in the "true" subgraph output by the stage two algorithm; they are moved to the join subgraph.

A discussion of the reasons for and against the creation of GOTOless programs can be found in an article by Knuth (1974). Although the inclusion of steps in the stage two control graph generation algorithm to create synthetic control variables and statements could be implemented, this has not been done in this research because such artificial modification of the program simply to eliminate explicit GOTOs does little to improve a human´s understanding of the program structure. Variables and tests are introduced that were not intended in the source program. A similar approach is taken to the problem of node splitting to achieve GOTOless code. It is not done although the control graph structure would make node splitting simply a matter of duplicating certain sequences of nodes and copying them into other parts of the graph.

Algorithm STAGE2. Input a stage one control graph with root x and output a stage two control graph.

1. If x is nil, return nil.
2. If x is a transfer node and the successor of x is "inline" following x, then return STAGE2 of successor of x.
3. If x is not a conditional node, return

$$\begin{tikzcd}
\textcircled{x} \arrow[d] \\
\boxed{\text{STAGE2 of successor of x}}
\end{tikzcd}$$

4. x must be a conditional node.

   Set A = STAGE2 of "true" subgraph of x.

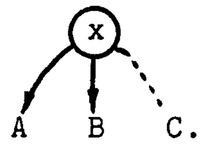
   Set B = STAGE2 of "false" subgraph of x.

   Set C = STAGE2 of "join" subgraph of x.

   (C will be nil if A and B are sink subgraphs.)

5. Set TARG = set of all join nodes in the graph A which are reached from the subgraph B. (Remember, all paths between A and B are directed from B to A.)

6. If TARG is empty, return



7. Find the join node in TARG which has the most graph structure beneath it. This is done by counting the number of nodes below the join node. Call this join node M.

8. Remove M from TARG.

9. Detach the subgraph in A rooted at M and attach it to the join subgraph, C. Adjust join links in A and B as necessary.

10. Go to step 6.

Step 2 causes a transfer node which has a successor immediately following it via a normal link (not a join or loop link) to be eliminated as being redundant. The heuristic used in step 7 to choose the subgraph which will become the join subgraph in the stage two representation is easy to compute and seems better than choosing at random. This causes the bulk of code that will appear in the join subgraph to be placed above the first label for an explicit jump from one of the other subgraphs.

CREATING THE INTERMEDIATE TEXT REPRESENTATION

After the stage two control graph has been generated, the decompiler associates with each node in the graph a statement or set of statements which express the semantics of the source instruction represented by that node. The decompiler translates the source instructions into intermediate instructions called IMTEXT (after Housel's name for his intermediate language). IMTEXT must cover the source language in the sense that all source language statements must have some IMTEXT representation. The details of the semantics of the source language are made explicit in the IMTEXT code in terms of IMTEXT semantics. Later analysis phases of the decompiler deal only with IMTEXT statements and so are decoupled from the particular source language input to the decompiler.

The IMTEXT Statement

A basic IMTEXT statement is of the general form of a quadruple:

(<loc>, <op>, <args>, <changes>)

where

<loc> indicates the position in the input text where the source instruction occurred, e.g., the address where the instruction begins;

<op> indicates the IMTEXT operator;

<args> is a list of arguments to <op>;

<changes> is a list of items changed by application of <op> to <args>.

For example, a translation of the program of Figure 1-A is given in Figure 4-E (the control structure is not shown).

The IMTEXT language is designed to be as simple as possible in form and semantics. It emphasizes the functional nature of the machine operations it represents -- some operation is applied to a set of operands and the result effects certain state changes to memory elements indicated in the <changes> list. The IMTEXT translation of each source statement is placed in its corresponding position in the stage two control graph as nodal information.

The first instruction of Figure 4-E, "load the A-register with the contents of memory location X," is translated to (0, MOVE, X, AREG). This translation attempts to make explicit the names of all variables participating in the movement of data. In this case the name AREG appears explicitly in the IMTEXT translation. No distinction is made between accumulators, index registers, or main memory elements in IMTEXT. All variables are treated exactly the same to avoid special case testing later on in the decompiler.

Addr		Source		IMTEXT
0		LDA	X	(0, MOVE, X, AREG)
1		ADD	=10	(1, ADD, (AREG, =10), AREG)
2		SUB	Y	(2, SUB, (AREG, Y), AREG)
3-4		JAZ	L1	(3.0, EQ, AREG, TMP)
				(3.1, BT, (TMP, L1), NIL)
5-6		JMPM	SUBR	(5, CALL, SUBR, SUBR)
7-8		JMP	L2	(7, B, L2, NIL)
9	L1:	LDA	J	(9.0, LABEL, L1, NIL)
				(9.1, MOVE, J, AREG)
10		ADD	=2	(10, ADD, (AREG, =2), AREG)
11		STA	J	(11, MOVE, AREG, J)
12	L2:	LDX	I	(12.0, LABEL, L2, NIL)
				(12.1, MOVE, I, XREG)
13		LDA	C(X)	(13.0, INDEX, (C, XREG), TMP)
				(13.1, MOVE, TMP, AREG)
14		SUB	J	(14, SUB, (AREG, J), AREG)
15		IXR		(15, ADD, (XREG, =1), XREG)
16		STA	B(X)	(16.0, INDEX, (B, XREG), STMP)
				(16.1, MOVE, AREG, STMP)
17		HLT		(17, HALT, NIL, NIL)

Figure 4-E. Source instructions of Figure 1-A, and their IMTEXT translations.

Translation to IMTEXT

For every instruction in the source program a sequence of corresponding IMTEXT instructions must be created. This mapping is provided by a set of functions which perform the following tasks:

1. recognize the machine operation;
2. select the appropriate IMTEXT translation routine for the operation;
3. form the argument and change lists from the explicit and implicit operands;
4. create synthetic variables and control mechanisms when necessary;
5. identify join nodes for the creation of IMTEXT LABEL statements; and
6. recognize idiomatic expressions.

Depending upon whether a symbolic source text is provided with the source program input to the decompiler, the recognition of the opcode can be directed toward the binary form or the symbolic form. A combination of both methods may help to disambiguate certain cases.

The mechanisms for selecting the appropriate IMTEXT translation routine is similar to that of any machine simulator -- the opcode is extracted from the instruction and used as an index into the proper routine through a branch table. The operation codes determine the interpretation of the rest of the instruction, i.e., which are the arguments and which are the targets for the changes effected by the operation.

The creation of synthetic variables for use in a one-to-many mapping of source code to IMTEXT is straightforward. Illustrations of this can be seen in Figure 4-E (statements 3, 13, 16) and Figure 4-F. The use of the synthetic variables TMP and STMP in the example corresponds to the use of temporary variables to hold the result of intermediate values while evaluating an arithmetic expression. These synthetic variables provide symbolic links between related IMTEXT expressions which can be used to create more deeply nested expressions during the expression analysis phase of decompilation. Synthetic variables may appear in the change list of an IMTEXT expression as well as in the argument list. (See statement 16 of Figure 4-E.) This facilitates handling of an indexed expression used to calculate a target address.

The presence of a join node in the control graph causes an IMTEXT LABEL statement to be generated. In addition, the source program symbolic labels are retained in the IMTEXT code. This provides for the inclusion of labels in the text of the output program if they are needed. For example, in Figure 4-E statements 9 and 12 include LABEL statements.

The recognition of idiomatic expressions can occur at the time that the source program is translated into IMTEXT. For simple idioms, this amounts to translating a sequence of source code as a contextual unit to one or more IMTEXT statements. A table of idioms and their translation can be kept in the decompiler, and a simple matching operation is performed to recognize the start of a potential idiom. The next few statements are examined to determined whether a complete idiom is present. Only the recognition of simple idioms is contemplated (non-looping, non-conditional, small number of statements) but such a facility is not implemented in the decompiler at this time. (See chapter 3 for some examples of idioms which might be processed in this manner.)

<u>Choice of IMTEXT Operations</u>

Some source instructions may require more than one IMTEXT statement to represent them. For example, the instruction "JAZ L1" of Figure 4-E means "if the contents of the A-register is zero, then jump to location L1." This has the IMTEXT translation of

```
(3.0, EQ, AREG, TMP)
(3.1, BT, (TMP, L1), NIL).
```

For simplicity, only atomic symbols (e.g. names, numbers) may appear in the argument or changes list of a basic IMTEXT statement. The instruction is translated in two parts. The first represents the calculation of a boolean value "Is AREG equal to zero?" and the result of the calculation is stored into a synthetic variable TMP. The second part of the instruction is an IMTEXT control statement which means "if the boolean value of the first argument (TMP) is true then branch to the location given by the second argument (L1)." No memory-elements are affected except the location counter. Since the location counter is affected by every instruction executed it does not appear on the change list of an IMTEXT statement.

One way to avoid generating multiple IMTEXT statements for one source statement is to define a new IMTEXT operator with the same meaning as the source statement. For example, we might define an opcode BZ in the form

(<loc>, BZ, (arg1, arg2), nil)

which would mean "if arg1 is zero then branch to arg2." But there are many conditions which could be tested in some machine languages. The design of the IMTEXT language must aim at some compromise between duplication of every possible instruction of the source language and a very simple language of primitive operations. The latter would result in frequent one-to-many mappings which would explode the size of the IMTEXT representation and lead to slow processing speeds during the analytic phase of the decompilation. (See Appendix I for a summary of IMTEXT statement types.)

A dramatic example of a source instruction which can balloon into many IMTEXT statements is shown in Figure 4-F. Up to nine conditions can be tested at one time by the jump instruction of the Varian 620/i computer. The conditions specified for testing must all be true (ANDed) in order to take the jump. These conditions are: "overflow" on (resets overflow if tested), A-register greater than or equal to zero, A-register less than zero, A-register equal to zero, B-register equal to zero, X-register equal to zero, sense switch 1 on, sense switch 2 on, and sense switch 3 on.

Figure 4-F illustrates the IMTEXT translation of a jump instruction which tests all nine conditions. Nineteen IMTEXT statements are created. Of course, this explosion of code is the exception rather than the rule. The great majority of jump instructions programmed for this machine test only one (or none) of the possible nine conditions. The 19:1 expansion ratio occurs because of the fact that there is no IMTEXT counterpart for a nine condition test. Since conditional tests are at the heart of program structure it is best that they not be obscured under machine dependent opcodes. For example, the IMTEXT statement

```
(0, BX, (777, ALPHA), NIL)
```

where BX is a new opcode which stands for the same thing as the Varian JMP instruction simply obscures what conditions are being tested.

```
Varian instruction : octal -- 001777, ALPHA

                     symbolic -- JIF 0777,ALPHA

IMTEXT representation:

        (0.1, MOVE, OF, T1)

        (0.2, MOVE, =0, OF)

        (0.3, GE, AREG, T2)

        (0.4, AND, (T1, T2), T1)

        (0.5, LT, AREG, T2)

        (0.6, AND, (T1, T2), T1)

        (0.7, EQ, AREG, T2)

        (0.8, AND, (T1, T2), T1)

        (0.9, EQ, BREG, T2)

        (0.10, AND, (T1,T2), T1)

        (0.11, EQ, XREG, T2)

        (0.12, AND, (T1, T2), T1)

        (0.13, MOVE, SS1, T2)

        (0.14, AND, (T1, T2), T1)

        (0.15, MOVE, SS2, T2)

        (0.16, AND, (T1, T2), T1)

        (0.17, MOVE, SS3, T2)

        (0.18, AND, (T1, T2), T1)

        (0.19, BT, (T1, ALPHA), NIL)
```

Figure 4-F. Example of the translation of one source instruction into many IMTEXT statements.

This causes the interpretation of the meaning of the instruction to be deferred until later in decompilation process. This example demonstrates that the IMTEXT translation must be carefully considered so that the proper mix between code volume and clarity of expression is achieved.

Machine Dependent Operations

Some source machine operations are difficult to generalize over a class of machines and so their representation in IMTEXT is best accomplished through the invocation of builtin subroutines. Instructions which fall into this class are shift operations, I/O operations, execute instructions, and operations on aggregates. Rather than breaking such operations into a sequence of simple IMTEXT instructions, a subroutine call might do very well as a representation if the analysis routines of the later decompiler stages will treat these operations as "black box" functions.

Side Effects

The effect that the execution of an instruction has upon the state of the machine can be decomposed into a primary effect and possible side effects.

The primary effect is usually described by the name of the instruction and involves either data movement or some calculation or both. For example, the primary effect of the instruction "ADD =10" in Figure 4-E is to add the value 10 to the A-register of the machine and store the resulting sum in the A-register.

The side effects produced by execution of an instruction are often associated with a condition code or status register which contains information about results of the operation performed. A side effect of the "ADD =10" operation is that the overflow flag of the machine is set if arithmetic overflow occurs.

Side effects can be classified into two sub-classes -- independent side effects and dependent side effects. The independent side effect always occurs when the operation is executed. The state of the memory element affected is always changed in some way independent of the previous state of that memory element. An example of this type of side effect is given by the IBM 360/370 instruction LTR "load and test register" which always changes bits 0-2 of the condition code to reflect whether the value loaded was equal, less than, or greater than zero. The state of the condition code bits 0-2 after execution of the LTR is only a function of the value loaded and is independent of the previous state of the condition code.

A dependent side effect may or may not cause the state of the affected memory location to be altered, depending on the previous state of the machine. The overflow indicator on most machines is an example of a memory element affected by dependent side effects of arithmetic instructions. If the overflow indicator is on and the machine instruction does not cause an overflow, then the indicator remains on. The value of the overflow indicator is not necessarily set or reset by an instruction such as "ADD =10." Its value might or might not be a function of the last instruction executed which could affect it.

The IMTEXT representation of an instruction provides for the representation of the memory elements of the machine affected by the execution of the instruction. The primary effect is of most interest -- the value of some memory element is changed. The side effects must be represented on the change list also. Representation of independent side effects is provided by listing the memory elements affected by the instruction on a sublist of the change list. Later analysis of instructions which require information about side effect state changes (such as the BC "branch on conditon" instruction of the 360/370) can refer to the last instruction which caused the side effect of interest. Representaton of dependent side effects produced by instructions does not provide much information of use to the decompiler because it cannot decide in general whether the side effect will actually be produced by a particular instruction without knowing the run-time variable history of the program.

The issue of including information about dependent side effects on the change list of an IMTEXT statement can be resolved in two ways:

1. provide a complete description of all memory elements of the machine which might possibly (but not always) be affected by the execution of an instruction. In this case a severe penalty is paid in our ability to analyze the positional relationships between instructions which have similar side effects; or
2. ignore the dependent side effects of instructions on the grounds that they are very infrequently used in most programs and admit that decompilations of programs which use such side effects may be in error due to invalid assumptions made about the usage history of variables.

For pragmatic reasons I have chosen to take the latter course. In a subsequent discussion of expression condensation and variable usage analysis, the reader will see examples of how dependent side effects can influence the creation of higher level IMTEXT constructs.

MORE COMPLEX IMTEXT STATEMENTS

This section deals with the algorithms for variable usage analysis and expression condensation which are used to create higher level IMTEXT statements. These new statements are constructed from the basic IMTEXT statements described previously. They represent groupings of arithmetic and logical statements into expressions. The process can be described as condensing a simple vertical non-nested computational description into a more complex horizontal nested description.

For example, the first three assembler language statements of Figure 4-E compute the value of one expression:

```
        0        LDA      X
        1        ADD      =10
        2        SUB      Y
represents       AREG := X + 10 - Y ;.
```

The IMTEXT representation of the assignment statement after expression condensation would be:

```
    (2,SUB,((1,ADD,((0,MOVE,X,AREG),=10),AREG),Y),AREG)
```

The intermediate stores to AREG can be removed because their only purpose is to save a temporary result for the next operation. The procedure for determining whether a variable use is a candidate for substitution or elimination is accomplished through variable usage analysis. This type of analysis is common to most optimizers and decompilers and is described in Aho and Ullman (1973), and Allen and Cocke (1976). Based upon the information gathered by the variable usage analysis algorithms applied to a section of the program, the combination of two or more IMTEXT statements may occur.

Expression Condensation

The method we use for expression condensation is called forward substitution. We will need a few definitions for the following discussion.

A variable V is used in an IMTEXT statement S if V appears on the argument list of S.

A variable V is changed in an IMTEXT statement S if V appears on the change list of S.

An IMTEXT control graph is a control graph with a statement or sequence of statements associated with each node of the graph in a natural manner -- each node represents a source program instruction or sequence of instructions translated into IMTEXT.

Given an IMTEXT control graph rooted at x, the value of a variable V entering the graph at x is needed in the graph if there exists a node y in the graph where V is used and there exists a path from x to y such that V is not changed at any node along that path.

The primary element of the change list of an IMTEXT statement represents that memory element which contains the primary value produced by the application of the IMTEXT operator to its arguments. The primary element of the change list may be nil, as in the case of a branch instruction.

Consider two statements S and T to be executed by machine M. Let (S, T) be a program (of two instructions) in which S is executed first, then T. Let (T, S) be a program in which T is executed first, then S. S and T are said to be _execution order independent_ if for all possible initial states of machine M, execution of the program (S, T) yields the same final state of M as execution of the program (T, S); otherwise they are _execution order dependent_.

Given two IMTEXT statements S and T, such that S precedes T in their IMTEXT control graph, the symbolic _forward substitution_ of S for V on the argument list of T means that S is removed from the graph and attached in place of V on the argument list of T.

Assumptions and Conditions for Forward Substitution

For forward substitution of statement S in statement T to maintain the functional meaning of a program, S must be execution order independent of every statement on every path from S to T in their control graph. This statement can be expressed in more detail by a number of assumptions and conditions necessary to guarantee that the meaning of a program is not changed by forward substitution of S in T.

Assumption 1. There exists a path from statement S to statement T in the IMTEXT control graph.

Assumption 2. S is not a transfer node and is not location dependent.

Assumption 3. All names in IMTEXT statements occurring on the paths(s) from S to T are unique. No two different variable names refer to the same memory element.

<u>Assumption 4</u>. There is only one primary element, V, on the change list of S. The primary effect of S is to change <u>one</u> memory element.

<u>Assumption 5</u>. V′ is a top level element of the argument list of T.

There are six conditions under which forward substitution can occur given the above assumptions. The above assumptions are also conditions which must be satisfied for forward substitution to occur. They are stated here as assumptions to simplify the following discussions. In the course of forward substitution the decompiler must also determine that the assumptions hold as well as the conditions stated below. In effect, the assumptions define S and T as <u>possible</u> candidates for forward substitution. The conditions examine the candidates (and the paths between them) more closely to determine if forward substitution may be accomplished. (In the following examples we use simple Algol-like statements rather than IMTEXT for clarity. The mapping between the two forms should be apparent to the reader.)

<u>Condition 1</u>. The symbol V is identical to the symbol V′. (We use the symbol V′ to distinguish the occurrence of V in

statement T. Later we will relax the statement of this condition so that V and V´ need not be identical symbols.)

Condition 2. No statement on the path(s) from S to T uses V as an argument or relies on any side effect of the evaluation of S. V does not appear on the argument list of any statement on the path.

An example of a failure of condition 2:

```
        S:      V := A + B;
        X:      C := V ;
        T:      D := V´ ;
```

Statement X requires V as an argument. The above sequence of statements cannot be transformed into

```
        X:      C := V ;
        T:      D := (V := A + B) ;
```

without changing the meaning of the program. The value of V to be stored into C is generated by the execution of statement S. Statement S must be executed before X.

Condition 3. No statement on the path(s) from S to T changes the value of V. V does not appear on the change list of any statement on the path.

An example of a failure of condition 3:

```
        S:      V := A + B ;
        X:      V := C ;
        T:      D := V´ ;
```

Statement X changes the value of V from that generated at S.

Statement T requires the V generated by X, not S, so the faulty substitution yielding

```
        X:      V := C ;
        T:      D := (V := A + B) ;
```

assigns the wrong value to D in statement T.

<u>Condition 4</u>. There are no statements on the path(s) from S to T that change any variables on the argument list of S. An example of failure of condition 4:

```
        S:      V := A + B ;
        X:      A := C ;
        T:      D := V´ ;
```

Statement X changes A which is an argument to S. The execution order of S and X is thus fixed. A substitution of S in T would lead to the sequence

```
        X:      A := C ;
        T:      D := (V := A + B) ;
```

The value of D is dependent on having the value of A (used to compute V) <u>before</u> X is executed, not after.

<u>Condition 5</u>. All paths from the root node of the control graph to T pass through S. Every path from S to T is a subpath of any path in the graph which includes T.
An example of a failure of condition 5:

```
        X0:     V := 0 ;
        X1:     IF Y = 0 THEN GOTO X3 ;
        S:      V := A + B ;
```

```
X2:    IF B > 0 THEN GOTO T ;
X3:    C := D ;
T:     E := V′ ;
```

There are three paths to T in the control graph rooted at X0 -- (X0, X1, X3, T), (X0, X1, S, X2, T) and (X0, X1, S, X2, X3, T). The value used for V in T is generated at X0 in the first path, and by S on the other two. A substitution of S in T yielding the sequence

```
X0:    V := 0 ;
X1:    IF Y = 0 THEN GOTO X3 ;
X2:    IF B > 0 THEN GOTO T ;
X3:    C := D ;
T:     E := (V := A + B) ;
```

would cause E to receive the wrong value when the path (X0, X1, X3, T) was executed.

Argument Evaluation Order

Arguments to basic IMTEXT statements are primitive items such as numbers, variable names, logical constants, and address constants. Once we allow the creation of more complex arguments by forward substitution we must be careful to consider the evaluation order of those arguments and be sure that order is reflected in the target language translation. For example, given the sequence

```
X:     V := 0 ;
S:     V := 1 ;
```

```
        T:      A := V + V ;
```

which is changed to

```
        X:      V := 0 ;
        T:      A := (V := 1) + V ;
```

by forward substitution, we can see that the operator ´+´ in T can no longer be considered commutative because the result of the evaluation of the arguments is order dependent. If the target language compiler permits undefined argument evaluation order (e.g., to achieve optimization), we must guard against violation of the following condition.

<u>Condition 6</u>. If there is more than one valid evaluation sequence of the argument list of an IMTEXT statement, the arguments must be evaluation order independent. (The definition of evaluation order independence is similar to that for execution order independence.)

The valid evaluation sequences allowed are a function of the interpreters of the IMTEXT statement or its isomorphic forms such as a higher level language statement. In the example above, a strict left-right order preserves the original meaning of the statement sequence while an undefined order does not. An example of the violation of this condition occurred while testing the decompiler. An oversight in the target code generation procedure caused an execution error in the target program. A two condition test was translated into an

```
IF c1 AND c2 THEN ...
```

form of target statement. Forward substitution of an assignment expression into c2 was done. In the MOL620 target language, the AND operator does not evaluate its second argument if the first one is false. When the target program executed and c1 was false, the substituted assignment in c2 was not executed, thus causing the error. The solution to this problem was to generate a call to a builtin function AND, e.g., AND(c1,c2), where the argument evaluation protocol insures that all arguments are evaluated.

<u>Explicit</u> <u>Assignment</u> <u>Elimination</u>

Assume statements S and T satisfy the above conditions for the forward substitution of S into the argument list of T as when

```
S:     V := A + B ;
T:     D := V′ ;
```

becomes

```
T:     D := (V := A + B) ;.
```

Under what conditions can we eliminate the intermediate storage into variable V and change statement T to

```
T:     D := A + B ;?
```

The only reason for changing the value of a memory element is to make that information available for later use. If it can be determined that the value to be stored in memory element V is not needed in V at some future time then the assignment can be eliminated. We add a new condition to the list.

Condition 7. Assume S has been substituted for V in the argument list of T. The explicit assignment of V in S can be eliminated (leaving just the calculation of the value) if there exists no statement X in the control graph rooted at T such that X uses V, or given that there is a statement X which uses V then every path (T, X) (exclusive of X itself), contains a statement which changes V.

An example of failure of condition 7:

```
        S:      V := A + B ;
        T:      D := V′ ;
        X:      E := V ;
```

The substitution with removal of the explicit assignment to V

```
        S:      D := A + B ;
        X:      E := V ;
```

is invalid because X needs V as an argument. An example where a valid substitution with removal of an explicit assignment is:

```
        S:      V := A + B ;
        T:      D := V′ ;
        X:      V := E ;
```

which can be changed to

```
T:      D := A + B ;
X:      V := E ; .
```

Of course, some higher level languages (e.g., FORTRAN) do not support assignment expressions but such a facility does improve the ability of the language to describe the storage of intermediate results involved in more complex calculations. If the higher level language does not support assignment expressions, then forward substitution is only fruitful when condition 7 is satisfied, i.e., when the explicit assignment can be removed.

Value Equivalence

A common sequence of code in an initialization section of a program is one which zeroes out a set of variables. For example,

```
1       A := 0 ;
2       B := 0 ;
3       C := 0 ;
```

After execution of statement 3, variables A, B, and C contain the same value, namely zero.

In assembler language programs multiple assignments are usually not expressible in one statement but in higher level languages it is a common syntactic feature. We would like to convert the above sequence of statements into a single statement of the form

```
A := B := C := 0 ;
```

if the syntax of the target language permits such a statement. Note, however, that the assumptions and conditions previously stated for forward substitution (our only method of creating complex expressions) prevent us from combining the statements of our example. Specifically, the change elements do not appear on any argument list so condition 1 does not hold. We need to expand our notion of symbolic equivalence to include the notion of value equivalence.

Two data items are said to be _value equivalent_ at node x in an IMTEXT control graph if they always contain the same value when execution (of the program described by the control graph) along any path to x has proceeded to and executed node x. In this context we consider a data item to be a variable or a constant.

The relation of value equivalence between two data items is naturally an equivalence relation in the mathematical sense of the term. The relation is _reflexive_ -- a data item is value equivalent to itself. The relation is _symmetric_ -- if a data item A is value equivalent to B (at node x) then B is value equivalent to A (at node x). The relation is _transitive_ -- if A is value equivalent to B and B is value equivalent to C, then A is value equivalent to C.

A _value equivalence list (VEL)_ relative to node x in an IMTEXT control graph is a list of elements each of which is a list of data items. All items on an element list are value equivalent at x.

To illustrate the notion of a value equivalence list, we provide the following example with the program statements on the left and corresponding value equivalence list on the right. We do not list simple reflexive elements (e.g., (A,A)) on the value equivalence list.

```
     1       A := 0 ;                  ((A,0))
     2       B := 0 ;                  ((A,0),(B,0)) =>
                                               ((A,B,0))
     3       C := 0 ;                  ((A,B,0),(C,0)) =>
                                               ((A,B,C,0))
     4       D := D + E ;              ((A,B,C,0))
     5       C := D ;                  ((A,B,0),(C,D))
     6       F := B + 1 ;              ((A,B,0),(C,D),(F,1))
```

At statements 2 and 3 the transitive nature of the value equivalence relation is used to combine elements of the value equivalence list. At statement 4 the value of E is undefined, so D is not entered on the VEL. Statement 5 causes the value of C to be changed. C is removed from its old VEL element and a new element (C,D) is created to show the value equivalence of C and D at statement 5. At statement 6 we see that the VEL can be used to determine a priori the value of F at statement 6. This is a feature of many optimizers -- the evaluation of constant expressions at compile time. In this case we know the value of B is zero when statement 6 is executed and the constant calculation (0+1) can be performed and the element (F,1) can be added to the VEL.

The notion of value equivalence can be used to extend the capability of a decompiler to perform symbolic substitution. Condition 1 can be relaxed so the V and V′ need only be value equivalent at T rather than being identical symbols. We state the new condition 1:

<u>Condition 1′</u>. V is value equivalent to V′ at T.

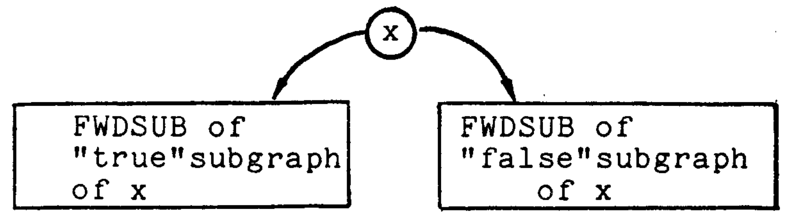
<u>Algorithmic Descriptions</u>

The descriptions of the more important algorithms involved in forward substitution are included in the following text. The calling order is:

1. FWDSUB -- given a control graph, returns with all substitutions made;
2. FWDSUB2 -- substitutes a root node in its subgraph;
3. FWDSUB3 -- substitutes root node into its son;
4. NEEDP -- determines whether a variable is "needed" in a graph;
   a. USEDP -- determines whether a variable is used in a graph;
   b. CHANGEDP -- determines whether a variable is changed in a graph.

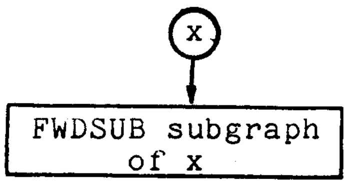
<u>Algorithm</u> <u>FWDSUB</u>. Given IMTEXT control graph X, perform forward substitution on X and return the altered graph. Let x be the root node of X.

1. If X has less than two nodes then return X. (There is nothing to substitute.)
2. If x is a conditional node then return



3. Perform substitution of x into X.

   Set X′ = FWDSUB2 of X.
4. If X′ is nil then x could not be substituted in its subgraph. Return



5. FWDSUB2 was successful in substituting x. Return FWDSUB of X′.

<u>Algorithm</u> <u>FWDSUB2</u>. Given an IMTEXT control graph X with root node x, attempt to perform forward substitution of x into X. Return the new graph if successful; otherwise return nil. Apply FWDSUB3 repeatedly on successive subgraphs of X.

1. Set B = nil.

   Set C = subgraph of x.

2. If the operator of x prohibits forward substitution (e.g., branch, halt, label, etc.) then return nil.
3. Set L to the graph returned by the application of FWDSUB3 to

$$\begin{tikzcd} \textcircled{x} \arrow[d] \\ C \end{tikzcd}$$

   FWDSUB3 also returns an indication of the success it had in trying to perform the substitution of x into C.
4. If "failure" is returned, then return nil.
5. If "success" is returned, then return the graph

$$\begin{tikzcd} B \arrow[d] \\ L \end{tikzcd}$$

6. FWDSUB3 must have returned "continue." We will bypass the root of C and try to substitute x further down in its subgraph. If the root of C is a conditional node then return nil. We do not try to do forward substitution along both subtrees. This assures condition 4 is satisfied.
7. Append the root of C to B.
8. Set C = subgraph of C.
9. Go to step 3.

Algorithm FWDSUB3. Given an IMTEXT control graph X with root node x, determine whether a forward substitution of x in its immediate successor is possible. Let y be the successor of x. (Note: x is not a conditional node.) Let V be the primary value of the change list of x.

1. Check condition 1. Does y use V (or any variable value equivalent to V) on its argument list? If so, then go to step 5.
2. Condition 1 failed. Shall we do symbolic substitution in the change list of y? Is the operation of x a member of a special group of operations allowed to be substituted in a change list (e.g., indexing or indirection on the left of an assignment)? If so, then substitute x for V on the change list of y. In this case, V is a synthetic variable and assignment elimination is guaranteed. Return the control graph rooted at y and the "success" indication.
3. Condition 1 failed and we did not substitute in the change list of y. The node y does not use V as an argument so condition 2 is not violated if x passes y. Does y alter the change list or argument list of x so that condition 3 or 4 is violated at y if we would try to substitute x past y? If so, then return "failure."
4. Return the "continue" indication, i.e., forward substitution of x did not succeed at y but y does not preclude a further attempt to substitute x in

the graph beyond y.

5. Condition 1 has been satisfied at y. Conditions 2, 3, and 4 are not relevant since y is the direct successor of x. Condition 5 is not relevant because we are guaranteed by higher level routines (FWDSUB and FWDSUB2) that the path (x,y) has only one entry, x. We check condition 6 (argument evaluation order independence) assuming a substitution of x in y. With an undefined evaluation order of arguments, then if any change lists of non-primitive arguments of y intersect with the argument lists or change list of x, then return "failure." If the change list of x intersects the argument list of y in more than one place, return "failure."
6. We must determine whether condition 7 holds so that the explicit assignment in x may be removed if possible. We compute whether the variable instances of the change list of x are needed in the subgraph of y. Set NEED = result of call to NEEDP with x and subgraph(s) of y as arguments.
7. Mark x as an explicit assigment to V if NEED is true (i.e., condition 7 does not hold) else mark as no assignment. Replace V with x in the argument list of y. Return the new graph rooted at y.

Several assumptions and shortcuts are reflected in the algorithms. We assume that any control graph being processed is completely specified -- all branches and joins are represented. This is valid only in the ideal case where the window on the program was large enough to encompass all of the code of an independent program segment and no asynchronous external entries are made into the routine. We will discuss more about this matter in a later chapter.

We assure that substitution is only done in straight-line code and stops at a fork (condition node) in the control graph. The problems of substituting in more than one subgraph at the same time are thus avoided.

Not all possible substitutions are performed. Second order substitutions are not attempted, as illustrated in the following example.

```
1       A := B ;
2       B := C ;
3       D := A ;
4       E := B ;
```

Statement 1 cannot be substituted in statement 3 because of statement 2 which changes the argument to statement 1 (i.e., B). The algorithm then attempts to substitute statement 2 in its subgraph. Statement 3 can be passed by 2 because no conditions are violated. We can substitute statement 2 in statement 4 and return the statement list

```
1       A := B ;
3       D := A ;
```

```
    4 .     E := (B := C) ;
```

Note that the substitution of statement 2 in statement 4 has now unblocked the possible substitution of statement 1 in statement 3. Another application of FWDSUB would yield

```
    3       D := (A := B) ;
    4       E := (B := C) ;
```

A change to algorithm FWDSUB3 could be made such that if condition 4 is violated we try to substitute the blocking statement further down the graph and thus remove the block.

<u>Algorithm</u> <u>FWDSUB3</u> <u>step</u> <u>3</u> <u>(revised)</u>.

3a. Condition 1 failed. Does y change the change list variables of x? If so then return "failure" because y must always block x from passing it in graph execution order.

3b. Does y change any of the arguments of x? If yes, then call FWDSUB2 of subgraph of x.

3c. If FWDSUB2 returned nil, then y could not be moved forward. Return "failure."

3d. If FWDSUB2 succeeded in returning a substitution graph, L, in which y has been moved forward, then return FWDSUB3 of

$$\begin{tikzcd} \textcircled{x} \arrow[d] \\ L \end{tikzcd}$$

Because of empirical observation of the utility of a revised algorithm, this change was rejected as having a low payoff for the extra computation. In addition, the unrevised algorithm prevents two substitutions in the same statement where they might be in conflict due to undefined argument evaluation order.

TRANSLATION TO THE TARGET LANGUAGE

After the substitution process described in the last section is complete, the next step in the decompilation involves translating the revised IMTEXT control graph to the higher level target language. We have two related problems:

1. translation of the IMTEXT statements themselves, and
2. translation of the control structure represented by the connections in the IMTEXT control graph.

IMTEXT Statement Translation

The decompiler must map every IMTEXT statement into a statement in the target language to achieve a complete translation. Of course, the IMTEXT statements generated are influenced by the target language. For example, unless imbedded assignment expressions are supported by the target language, the analysis needed to generate such statements is unnecessary. The use of builtin function calls to represent operations not supported in the target language provides escape from the target language. In a similar fashion, the use of calls to assembler language routines from FORTRAN programs provides the same escape mechanism.

Associated with each IMTEXT operator is a pattern which determines the target language text to be generated for the operator as applied to its operands. A procedure recursively evaluates the arguments to the IMTEXT operation, if necessary. This yields translations of non-primitive subexpressions built up in the expression condensation analysis.

Some IMTEXT opcodes are "top-level" operations in that they do not occur inside expressions. Examples of this type of IMTEXT opcode are CALL, GOTO, HALT, and LABEL. Other IMTEXT opcodes may appear inside expressions. Examples of these are MOVE, ADD, and SUB, i.e., those corresponding to the primitive expression operators in most high level languages.

Assume we wish to translate (out of context) an IMTEXT statement from the control graph. Algorithm NODEGEN describes the general process of translation. It uses the IMTEXT opcode of the statement to select a list of elements called a generation pattern. The elements of the pattern are of two kinds:

1. primitive elements such as numbers, arithmetic operators, and other strings, which require no further evaluation, and
2. function calls which are evaluated. (The result of the function evaluation is a primitive element.)

Some examples of top-level generation patterns for a language like Algol, given the IMTEXT statements, are:

IMTEXT	pattern
(-,ADD,(arg1,arg2),target)	#<target>:= #<arg1> + #<arg2>
(-,SUB,(arg1,arg2),target)	#<target>:= #<arg1> - #<arg2>
(-,LABEL,arg)	arg :
(-,CALL,arg)	CALL arg
(-,BUMP,arg,target)	#<target> := #<arg> + 1
(-,MOVE,arg,target)	#<target> := #<arg>

In the above patterns, the primitive elements are CALL, :=, +, -, :, and 1. The non-primitive elements are #<target>, #<arg>, #<arg1>, and #<arg2>. The general meaning of the function calls of the form #<x> is "evaluate the item (selected by) x" and each is strictly defined by its procedural definition in the decompiler. The majority of these non-primitive items select a portion of the IMTEXT statement and recursively call NODEGEN.

Argument level generation patterns for a language like Algol are similar to those for top-level statements except no assignment is made to the change list variables except for the MOVEXP ("move explicit") opcode. IMTEXT statements at the argument level got there by expression condensation. The explicit assignment is indicated by imbedding an IMTEXT statement in a MOVEXP statement. Some sample argument level patterns are given below:

IMTEXT	pattern
(-,ADD,(arg1,arg2),-)	(#<arg1> + #<arg2>)
(-,SUB,(arg1,arg2),-)	(#<arg1> - #<arg2>)
(-,MOVE,arg,-)	#<arg>
(-,MOVEXP,arg,target)	(#<target> := #<arg>)

Note that for nested expressions it is necessary to indicate in the target language the order of evaluation of the subexpressions. Parentheses in the pattern serve this purpose.

Algorithm NODEGEN. Given an IMTEXT statement, x, generate a target language statement. On entry the output list is empty.

1. If x is an argument to another higher level IMTEXT statement, retrieve the argument generation pattern using the opcode of x as a selector, otherwise select the top-level evaluation pattern for x.
2. For all elements in the generation pattern:
   a. if element is primitive, add it to the output list, or
   b. if element is non-primitive, evaluate it and add the result to the output list.
3. Return the output list.

Note that NODEGEN is independent of the target language of the decompiler. The generation patterns and any functions necessary to implement the non-primitive elements are target language dependent; however, the patterns will be very similar over a wide variety of target languages.

IMTEXT Control Structure Translation

The edges in an IMTEXT control graph represent the execution paths between nodes. Since one of the goals of this dissertation research is to create higher level control structures as well as higher level statements, part of the task of translating to the target language must include an analysis of the possible paths through the graph to discover control patterns which can be represented in the target language.

We can consider this a pattern recognition problem. Given the target language, we can define the primitive control patterns implemented by its higher level control statements. Examples of such statements are IF ... THEN ... ELSE, WHILE, REPEAT, CASE, and FOR. Let us define this set of target patterns as

$$P = \{ p[i] \mid i\text{=index of pattern} \}.$$

We can approach the problem of finding elements of P in the control graph in two ways:

1. Find pattern q in the control graph X such that q exists in P, or
2. Find a transformation t on the control graph X such that t preserves the logical execution equivalence of the program and t(X) contains pattern q exists

in P.

We define a relation (<) on the patterns of P such that if p[i] < p[j] then pattern p[i] is found in pattern p[j]. In terms of programming languages, the pattern of an IF statement is found in the patterns for conditional iterative statements. The pattern for a WHILE statement is found in a FOR statement. We have

p[IF] < p[WHILE] < p[FOR].

In decompilation it is usually desirable to try to find the most complex pattern in a control graph. Therefore, the order in which we attempt pattern matching should be governed by the inclusion relationships of possible patterns.

Associated with each pattern in P is a procedure which will attempt to find that pattern in the IMTEXT control graph and will generate target code based upon the mapping of that pattern into the control statements of the target language. The procedures for finding patterns and generating code are mutually recursive so the subpatterns are naturally processed. The pattern matching procedure looks for characteristics in the control graph under examination. The starting characteristic of a pattern is a top level coarse discriminator of what subgraphs might possibly fit the pattern. For example, the starting characteristic of a simple loop pattern is the fact that the first node in the pattern is a join node.

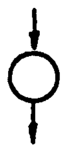
Algorithm STRUCGEN describes the general process of translating an IMTEXT control graph by successive applications of pattern processors.

<u>Algorithm</u> <u>STRUCGEN</u>. Examine the IMTEXT control graph X for patterns in P and return the target language translation of X.

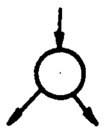
1. If X is empty, then return nil.
2. Set Q = the set of patterns we are looking for, i.e., P.
3. Select pattern(s) from Q such that the starting characteristic of the pattern match in X. The most complex pattern is tried first. Call this pattern q.
4. Apply the pattern procedure for q to X.
5. If q failed, then remove q from Q and go to step 3. (Note that the simplest pattern of P will always succeed. Its pattern procedure is NODEGEN. This fact assures that every X has a translation.)
6. The processor for q succeeded. The pattern processor returns X′, the subgraph of X not translated (the pattern of q may only have been a subpart of X), and T the target translation of the pattern q matched. We translate the rest of X. Return the append of T with STRUCGEN of X′.

Assume the basic control graph patterns of the target language can be represented by P = {p[1], p[2], p[3], p[4]}, where

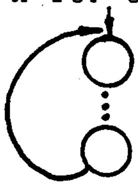
p[1] is a simple statement, single node, one entry, one exit.



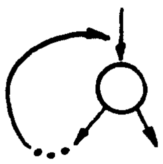
p[2] is a pattern for a conditional test (IF ... THEN)



p[3] is a pattern for a simple loop (DO FOREVER...)



p[4] is a pattern for a loop with a pretest (WHILE ... DO ...)



The starting characteristic of p[3] and p[4] is that the first node of the pattern is a join node. A distinguishing characteristic between p[3] and p[4] is that the first node in p[4] is also a conditional node. Note the p[3] < p[4] implies that p[4] will be tried first by STRUCGEN when the starting characteristic in a control graph is a join node and thus the patterns p[3] and p[4] are selected.

We describe the basic mechanisms of a pattern procedure for p[4] -- a loop with pretest.

Algorithm WHILE-DO. Matches pattern corresponding to a loop with a pretest in a control graph X whose root node, x, is a join node.

1. Is the join (root) node also a conditional node? If no, return "failure."

2. The join node is a conditional node. Search down the "true" subgraph looking for a transfer node branching back to x. If not found, then go to step 4.

3. The "true" subgraph loops back to x. Generate the code

   ```
   WHILE NODEGEN of argument of x
   DO STRUCGEN of "true" subgraph of x ;
   STRUCGEN of "false" subgraph of x.
   ```

   This code is returned; It represents the complete translation of X.

4. The "true" subgraph did not loop back. Try the "false" subgraph. Search the "false" subgraph of x for a transfer node branching to x. If not found, then return "failure." The transfer node to x did not exist in the graph X. The transfer to the join node x must be from outside of X.

5. The "false" subgraph loops back. Generate and return the code

```
        WHILE negate of NODEGEN of argument of x
        DO STRUCGEN of "false" subgraph of x ;
        STRUCGEN of "true" subgraph of x .
```

Note that the condition represented in the IMTEXT statement of x must be negated since the "false" subgraph loops while the expression is not true.

There are many patterns which would occur in a source program control graph which are not in P. We can define a set of transformations, T, which can be applied to these source program patterns and yield patterns in P. These transformations should be applied to the IMTEXT control graph before a search for the patterns of P is attempted. Examples of such transformations are node splitting and introduction of synthetic variables. The approach taken in this experiment has been that such transformations yield code and structure (by the process of node splitting or synthetic variable creation) which is not present in the original program.

POSTPROCESSING

After the entire source program has been decompiled, the text may then be subject to two phases of postprocessing:

1. automatic -- this involves modifying the output of the code generator by running a program that will
    a. create readable indented code structure,

b. place comments in appropriate positions in the output listing to improve readability,

c. transliterate symbol strings for readability or compatibilty with the uses of the text;

2. manual -- a human examines the code and modifies it to

   a. eliminate self-modifying code (marked as such by the decompiler),

   b. adjust the code to account for information lost due to inadequate window size,

   c. correct any syntactical errors which may have been created in the decompiled code.

The need for the postprocessor is purely pragmatic. It cleans up details not considered in the code generation of the decompiler. All changes made by the automatic processor are low level formatting or naming considerations. The manual processing involves some understanding of the source program. Together these two phases of the postprocessor change the program for readability (by formatting), compilabilty (by removing syntax errors), and executability (by adding or changing code).

In any decompilation system the manual intervention necessary should be kept to a minimum but it is not practical in most situations to entirely remove the need for manual editing because of the difficulties of implementing these manual changes in an automatic program. An example of the manual interventions required to decompile a substantial program will be given in a following chapter.

SUMMARY

This chapter has described a design for an experimental decompiler. The following topics have been discussed in detail: preparation of the input source, loading the prepared input into the decompiler, translation to a control graph structure and intermediate language for further analysis, condensation of the graph structure, pattern matching and generation of the target code, and postprocessing of the target program. The schematic diagram for the information flow through this system is shown in Figure 4-G.

The following chapter (5) describes an implementation of this design. Specific source and target languages are chosen and a small program is walked through the decompiler step by step to show the reader a more concrete example. Chapter 6 describes a substantial decompilation experiment and discusses the results of the experiment in relation to other results in the literature.

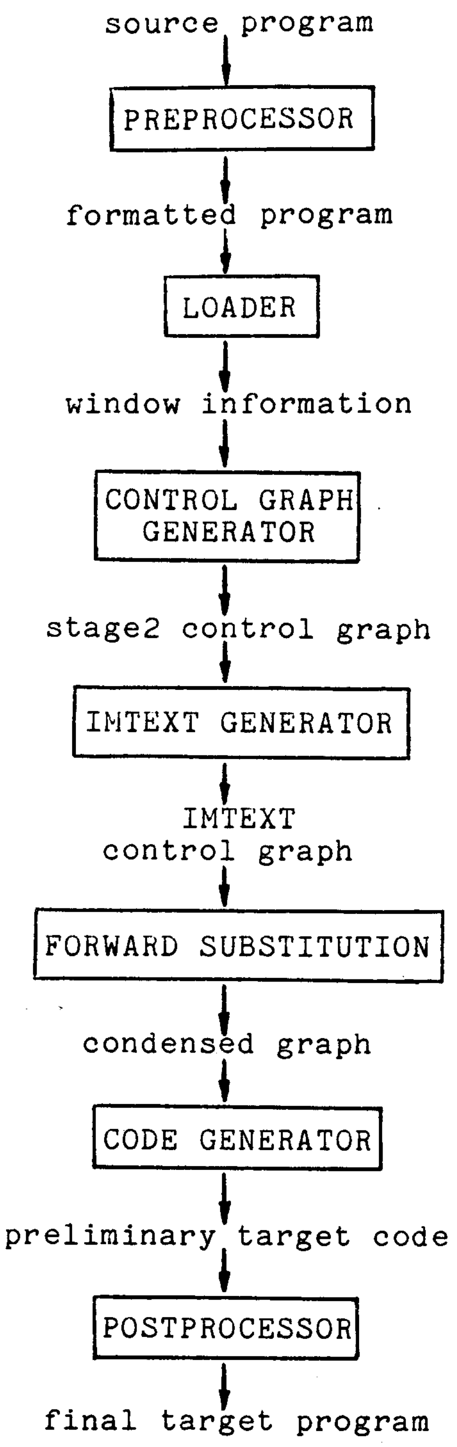


Figure 4-G. Information flow through the decompiler.

Chapter 5

A DECOMPILER IMPLEMENTATION

INTRODUCTION

Some of the decompiler design hypotheses presented in earlier chapters have been tested in an experimental decompiler system. The input text is a program (PL) written in a language which compiles into code for the target machine. The target language program (PH) will also compile into code for the target machine. This approach provides the opportunity for direct comparison of PL and PH in terms of object program size, code efficiency, and understandability and avoids questions that might arise due to differences in machine architecture.

CHOOSING THE SOURCE AND TARGET LANGUAGES

The source and target languages chosen for the initial decompilation tests are languages for the Varian Data Machines 620/i (16-bit) minicomputer. The source language is the assembler and machine language for the 620/i (Varian Data Machines 1968). See Appendix II for a summary of the 620/i instruction set. The target language is MOL620, a machine oriented language for the 620/i developed at U.C. Irvine (Hopwood and Hopwood 1971). See Appendix III for a summary of the MOL620 language.

CHOOSING A DECOMPILER IMPLEMENTATION LANGUAGE

The decompiler itself was written in UCI LISP (Weissman 1967, Bobrow et.al. 1973) and runs on the DECsystem-10 computer. The LISP system was chosen because it is interactive, accessible from a large number of terminals, and provides for the easy construction and manipulation of the complex data structures necessary for the decompilation processes. UCI LISP provides the flexibility of an interpreter during debugging but also includes a compiler that can be used to achieve fast execution speeds and reduce the cost of running the decompiler in a production mode.

The preprocessor and postprocessor were written in SNOBOL (Griswold et.al. 1971) for ease of character manipulation.

SIZE OF THE DECOMPILER

In order to give the reader some idea of the amount of code in the decompiler, I have attempted to define a measure of the size of each section.

The computation of the <u>volume</u> of the SNOBOL programs (the pre- and post-processors) uses a typical metric. (Since these programs are relatively small compared to the rest of the decompiler any reasonable metric will do.) We define the volume of a SNOBOL program to be the number of lines of code in the program.

It does not make much sense to count the number of lines of code in a LISP program, since LISP is a free format language where the notion of a physical line of code is meaningless. Counting the number of non-blank characters in a LISP s-expression (a function is an s-expression) might be a better metric but that method is sensitive to the programmer's choice of symbolic names, e.g., a ten character name would have ten times the mass of a one character name. Similarly, a count of the number of functions defined in each section of the decompiler is not a good measure of the amount of code either.

We define another measure which does a better job of indicating the number of logical units in a LISP s-expression. All syntactic objects in LISP are either atoms or lists, therefore we define the <u>mass</u> of a LISP function to be

mass = # of atoms + # of lists.

For example the mass of the s-expression

```
(A B C (D (E F) G) H (I))
```

is thirteen (13). There are nine atoms and four lists in the expression.

Table 5-A presents a summary of the mass of each section of the decompiler. The percentages in the section on the LISP programs were calculated by dividing the mass of the particular part of the program by the total mass of the LISP code in the decompiler.

Table 5-A. Sizes of various parts of the decompiler.

SNOBOL programs

Preprocessor	# of code lines = 86;
Postprocessor	# of code lines = 187

LISP programs

	# of functions	mass	%
Driver -- sequences execution of other parts of decompiler.	17	1190	10
Loader -- loads the window.	4	434	4
Stage1 Trace -- creates stage one control graph.	13	845	7
Stage2 Trace -- creates stage two control graph.	9	780	6
IMTEXT Generation -- creates IMTEXT nodes in graph.	33	2667	22
Forward Substitution -- performs forward substitution in IMTEXT graph	16	1389	11
Code Generation -- target code is generated from IMTEXT graph.	44	3620	29
Other -- service routines used by more than one of the above sections.	40	1402	11
Total -- sum of above	176	12327	100

The decompiler occupies 28K 36-bit PDP-10 words, not including the LISP interpreter itself (15K) or any dynamic storage. Dynamic storage includes push-down stacks for recursive function calls and storage for the information in the decompiler window. The sample program presented in this chapter was decompiled using 44K words of memory including the LISP interpreter.

A SAMPLE PROGRAM TO BE DECOMPILED

The descriptive approach taken in this chapter will be to show the reader the various stages in the decompilation of a small segment of source code. A definition of the algorithm used in the example follows.

<u>Algorithm</u> <u>MOVE</u>. Given three global arguments

P1 = starting address, source

P2 = ending address, source

P3 = starting address, target

move the contents of memory locations [P1,P2] to the locations beginning at P3. We cannot move data to locations higher than the address in variable H.

1. Exchange P1 and P2 if P1 > P2. (Call PCHK.)
2. If the address of the last target location (P3+P2-P1) is above H then load an error code ('?H') and go to the error routine (ERR).
3. Move the data.
4. Return.

Figure 5-A is the assembler output listing of the MOVE subroutine written in 620/i assembler language (DAS). The first column of decimal numbers contains the sequence number of the source line. The second column contains the octal addresses where the code is generated. The third column of octal numbers is the machine code itself. The rest of the line is the source statement -- label, opcode, operand(s), and comments field. The meanings of the opcodes of Figure 5-A are given in Figure 5-B.

This sample program has been written to illustrate the action of the decompiler. While its complexity is limited for simplicity of description, it does have some interesting features.

PCHK (lines 5 and 14) is an external subroutine. All variables are considered global. The EQU statement (line 5) simply gives a definition to the symbol PCHK for the purposes of assembling this program fragment out of context.

ERR (lines 6 and 22) is an external error routine. It is given an arbitrary definition at line 6.

Lines 20 and 31 contain jump instructions using location counter expressions for the jump address target instead of a label. This practice is common in many assembler language programs. The decompiler will use the address field of the machine code to determine the target of the jump.

Since there is no "jump if A-register less than or equal to zero" instruction on the 620/i an "identity" frequently used to accomplish this test is

$$A <= 0 \text{ implies } A - 1 < 0$$

(See lines 19-20.) There is a similar code sequence on lines 25-26 for testing whether the A-register is strictly greater than zero.

The instructions sequences mentioned above (lines 19-20 and 25-26) are examples of idiomatic sequences which are soft implementation characteristics (see Chapter 3) of programs written for the 620/i. However, the "identity" used fails when the contents of the A-register before the DAR instruction is the largest negative number representable on the machine, -32768. Subtracting one from this number yields a positive number (32767) with overflow.

Identities of this type are used because they work for <u>most</u> of the domain of variables to be tested. Extending the code sequence of the test by adding another check for overflow is almost never done because the extra code needed would rarely be used. The programmer either forgets about the possibility that the sequence might fail, realizes it might fail but codes it anyway, or proves to himself that the domain of the test does not ever include -32768 (although he may not be able to guarantee this fact for all future revisions of the program).

The lack of complete instruction sets creates these pitfalls; many programmers stumble into them sooner or later as they try to code around missing instructions. The problem is particularly acute on minicomputers where the size of the instruction word is limited and designers cannot put all of the instructions in the architecture which would be desirable.

```
 *
 *      SAMPLE PROGRAM TO DECOMPILE
 *
       000200      ,ORG   ,0200   FIRST ADDRESS OF PGM
       005000 PCHK,EQU   ,05000  EXTERNAL SUBROUTINE
       006000 ERR ,EQU   ,06000  EXTERNAL ERROR ROUTINE
 *
 *MOVE - M <START.SOURCE> <END.SOURCE> <START.TARGET>
 *      MOVE WORDS [P1,P2] TO [P3,P3+P2-P1]
 *      MUST HAVE P1<=P2, P2<=H, L<=P1, P3+P2-P1<=H
 *      ELSE GOTO ERROR ROUTINE
 *
000200 000000 MOVE,ENTR  ,       ENTRY POINT
000201 002000     ,CALL  ,PCHK   CHECK PARMS P1,P2
000202 005000
000203 010232     ,LDA   ,P2
000204 120233     ,ADD   ,P3
000205 140231     ,SUB   ,P1
000206 140234     ,SUB   ,H
000207 005311     ,DAR   ,
000210 001004     ,JAN   ,*+6    (FINAL TARGET > H; ERROR.
000211 000216
000212 006010     ,LDAI  ,´?H´   ERROR CODE
000213 137710
000214 001000     ,JMP   ,ERR    TO ERROR ROUTINE
000215 006000
000216 010231     ,LDA   ,P1     LOOP HEAD
000217 140232     ,SUB   ,P2     CHECK TERMINATION
000220 005311     ,DAR   ,
000221 001002     ,JAP*  ,MOVE   DONE
000222 100200
000223 017231     ,LDA*  ,P1     GET SOURCE WORD
000224 057233     ,STA*  ,P3     STORE IT
000225 040231     ,INR   ,P1     BUMP SOURCE PTR
000226 040233     ,INR   ,P3
000227 001000     ,JMP   ,*-9    LOOP AGAIN
000230 000216
 *
000231 000000 P1  ,DATA  ,0
000232 000000 P2  ,DATA  ,0
000233 000000 P3  ,DATA  ,0
000234 000000 H   ,DATA  ,0
       000000     ,END   ,
```

Figure 5-A. Sample source program to be decompiled.

ADD --	add memory to A-register; result to A.
CALL--	subroutine call. (See ENTR).
DAR --	decrement A-register.
DATA--	declares symbol and presets memory value.
END --	end of assembler program.
ENTR--	entry point; return address put here by a CALL.
EQU --	gives definition to symbol.
INR --	increment memory.
JAN --	jump if A-register negative (< 0).
JAP*--	jump indirect if A-register positive (>= 0).
JMP --	unconditional jump.
LDA --	load A-register.
LDA*--	load A-register indirect.
LDAI--	load A-register immediate.
ORG --	sets origin address of program.
STA*--	store A-register indirect.
SUB --	subtract memory from A-register; result to A.

Figure 5-B. Definition of sample program opcodes.

A decompiler has several options available to it when translating these idiomatic sequences

- o flag the sequence as "dangerous";
- o assume the identity is true and use it to create the higher level code; or
- o translate the code sequence without use of the identity.

The experimental decompiler described here tries to use the identity to simplify target expressions.

A more complex transformation of the source code from a "natural" statement of the algorithm into an "efficient" one is harder for a decompiler to recognize. For example suppose that the source machine can only test the value of an index register for zero (as is the case on the 620/i). A common approach to coding iterative loops which modify an indexed vector is to count down from the maximum index to zero, or start with a negative index and count up to zero. The problem for a decompiler is to recognize whether the order of access to the vector is important or is the loop simply a statement of "for all elements of the vector do something." If the latter interpretation is the case, then the details of indexing and accessing could be suppressed in the target program under a suitable high level statement.

On lines 25-26 we see the use of the "identity"

$$A > 0 \text{ implies } A - 1 >= 0.$$

In this test for loop termination, we could avoid executing the DAR instruction every time through the loop by incrementing P2 before the loop began and the test for completion could be

$$P1 >= P2$$

instead of

$$P1 > P2.$$

This is another transformation of the "natural" code sequence which yields more efficient execution but obscures the real intent of the code. As modifications of the program to achieve efficiency are carried out on a more global scale, they are more difficult for the decompiler (or a human) to relate to their purpose (e.g. test for loop termination) and are simply translated without modification to the target program.

```
(0. NIL NIL NIL () "%*%" (  ))
(1.     NIL     NIL     NIL     ()      "%*%"   ())
(2.     NIL     NIL     NIL     ()      "%* SAMPLE PROGRAM TO DECOMPILE%"       ())
(3.     NIL     NIL     NIL     ()      "%*%"   ())
(4.     NIL     NIL     ORG     (0200)  "%FIRST ADDRESS OF PGM%"        (000200))
(5.     NIL     PCHK    EQU     (05000) "%EXTERNAL SUBROUTINE%" (005000))
(6.     NIL     ERR     EQU     (06000) "%EXTERNAL ERROR ROUTINE%"      (006000))
(7.     NIL     NIL     NIL     ()      "%*%"   ())
(8.     NIL     NIL     NIL     ()      "%*MOVE - M <START\056SOURCE> <END\056SOURCE> <START\056TARGET>%" ())
(9.     NIL     NIL     NIL     ()      "%* MOVE WORDS [P1,P2] TO [P3,P3+P2-P1]%"       ())
(10.    NIL     NIL     NIL     ()      "%* MUST HAVE P1<=P2, P2<=H, L<=P1, P3+P2-P1<=H%"       ())
(11.    NIL     NIL     NIL     ()      "%* ELSE GOTO TO ERROR ROUTINE%"        ())
(12.    NIL     NIL     NIL     ()      "%*%"   ())
(13.    000200  MOVE    ENTR    ()      "%ENTRY POINT%" (000000))
(14.    000201  NIL     CALL    (PCHK)  "%CHECK PARMS P1,P2%"   (002000 005000))
(15.    000203  NIL     LDA     (P2)    NIL     (010232))
(16.    000204  NIL     ADD     (P3)    NIL     (120233))
(17.    000205  NIL     SUB     (P1)    NIL     (140231))
(18.    000206  NIL     SUB     (H)     NIL     (140234))
(19.    000207  NIL     DAR     ()      NIL     (005311))
(20.    000210  NIL     JAN     ("*+6") "%\050FINAL TARGET\051 > H\073 ERROR\056%"      (001004 000216))
(21.    000212  NIL     LDAI    ("'?H'")        "%ERROR CODE%"  (006010 137"10))
(22.    000214  NIL     JMP     (ERR)   "%TO ERROR ROUTINE%"    (001000 006000))
(23.    000216  NIL     LDA     (P1)    "%LOOP HEAD%"   (010231))
(24.    000217  NIL     SUB     (P2)    "%CHECK TERMINATION%"   (140232))
(25.    000220  NIL     DAR     ()      NIL     (005311))
(26.    000221  NIL     JAP*    (MOVE)  "%DONE%"        (001002 100200))
(27.    000223  NIL     LDA*    (P1)    "%GET SOURCE WORD%"     (017231))
(28.    000224  NIL     STA*    (P3)    "%STORE IT%"    (057233))
(29.    000225  NIL     INR     (P1)    "%BUMP SOURCE PTR%"     (040231))
(30.    000226  NIL     INR     (P3)    NIL     (040233))
(31.    000227  NIL     JMP     ("*-9") "%LOOP AGAIN%"  (001000 000216))
(32.    NIL     NIL     NIL     ()      "%*%"   ())
(33.    000231  P1      DATA    (0)     NIL     (000000))
(34.    000232  P2      DATA    (0)     NIL     (000000))
(35.    000233  P3      DATA    (0)     NIL     (000000))
(36.    000234  H       DATA    (0)     NIL     (000000))
(37.    NIL     NIL     END     ()      NIL     (000000))
```

Figure 5-C. Output of the preprocessor for sample program.

PREPARATION OF THE SOURCE PROGRAM

The source program of Figure 5-A is input to the preprocessor for formatting as described in the previous chapter. The preprocessor generates a sequence of LISP s-expressions where each s-expression corresponds to a line of the source listing. The format of the s-expression generated is

```
( <sequence number>
  <location address>
  <label>
  <opcode>
  <operands list>
  <comments>
  <value list> ).
```

The item NIL or ´()´ on a list means that field is empty. Figure 5-C shows the output of the preprocessor.

The source input listing is read from a disk file and the output goes to a disk file. The preprocessor is a one-pass program and handles one line of input at a time. Its run-time memory requirement is independent of the length of the source program. Its execution time is proportional to the length of the input program.

Some remarks about Figure 5-C -- line 0 is a dummy inserted into the code by the preprocessor. There are certain characters which are difficult to handle in LISP because they have reserved meanings, e.g., ´(´, ´)´, ´.´;

these are replaced with a coded representation which consists of their three digit ASCII code prefixed with a back-slash ('\'). (See lines 8 and 20.) They are changed back by the postprocessor.

```
(006000 . ERR)
(000234 . H)
(000200 . MOVE)
(000231 . P1)
(000232 . P2)
(000233 . P3)
(005000 . PCHK)
```

Figure 5-D. Symbol table entries for sample program.

LOADING THE FORMATTED CODE

The preprocessor extracts the symbol table from the assembler listing (see Figure 5-D). The symbol table is input to the decompiler and then the source file created by the preprocessor (see Figure 5-C) is loaded into the decompiler window. The window corresponds to the M and S vectors described in Chapter 4. (The reader may wish to refer to Figure 4-A.) In the case of this sample program, the loader stops when it sees the end-of-file. Some source lines in S are not pointed to from M. These correspond to lines which do not generate any code, e.g., lines 1-12, 32, 37.

CONTROL GRAPH GENERATION

Once the window is loaded, the decompiler driver passes control to the stage one control graph generation function.

That function will attempt to generate a control graph rooted in the M vector at M[0] (M[0] will correspond to program location 200). Figure 5-E shows a graphical representation and the internal LISP representation of the stage one control graph. After this graph is generated, the driver invokes the stage two function which will attempt to indicate join nodes of the graph explicitly. Figure 5-F illustrates the stage two graph.

Stage One

The control graph generation algorithm looks for the first executable memory word in the M vector. This is at location 201. The ENTR pseudo-op indicates that location 200 is not an instruction. Since location 201 is a CALL instruction, the stage one algorithm seeks information regarding the attributes of this subroutine from a service function. If this information is not present in the data available to the decompiler, the service routine asks the user for the information. In this case a short dialog between the decompiler and the user takes place.

decompiler: "How many in-line arguments to subroutine PCHK called from 201?"

user: "0"

decompiler: "Does it return after the call?"

user: "yes"

These are the only two subroutine attributes the decompiler is programmed to request. It remembers the response to the questions in case it might need the information later. The number of in-line arguments enables it to determine where control will commence after the CALL returns, if it does return. The procedures used to determine subroutine attributes may be as complicated as necessary. For example, if all argument lists are in-line and the first word of the list is a constant count of the number of words on the list, a small procedure could be programmed to extract this information from the source program itself without asking the user.

The stage one algorithm determines the successor of the instruction at 201 is at 203. It continues interpreting the control flow until it gets to the conditional instruction at 210. The control flow splits here -- one path goes to the error routine (ERR), the other continues. By convention, the "true" path (to location 216) is traced first. Because we have the machine code present, we do not have to determine what "*+6" means. Its value is the second word of the "JAN" instruction which indicates the target of the jump is 216. By having the machine code, we avoid having to assemble the source code again.

```
(201
 203
 204
 205      LISP s-expression representation
 206
 207
 210
 (216 217 220 221 (nil) (223 224 225 226 227 + * 216))
 (212 214 + * 6000))
```

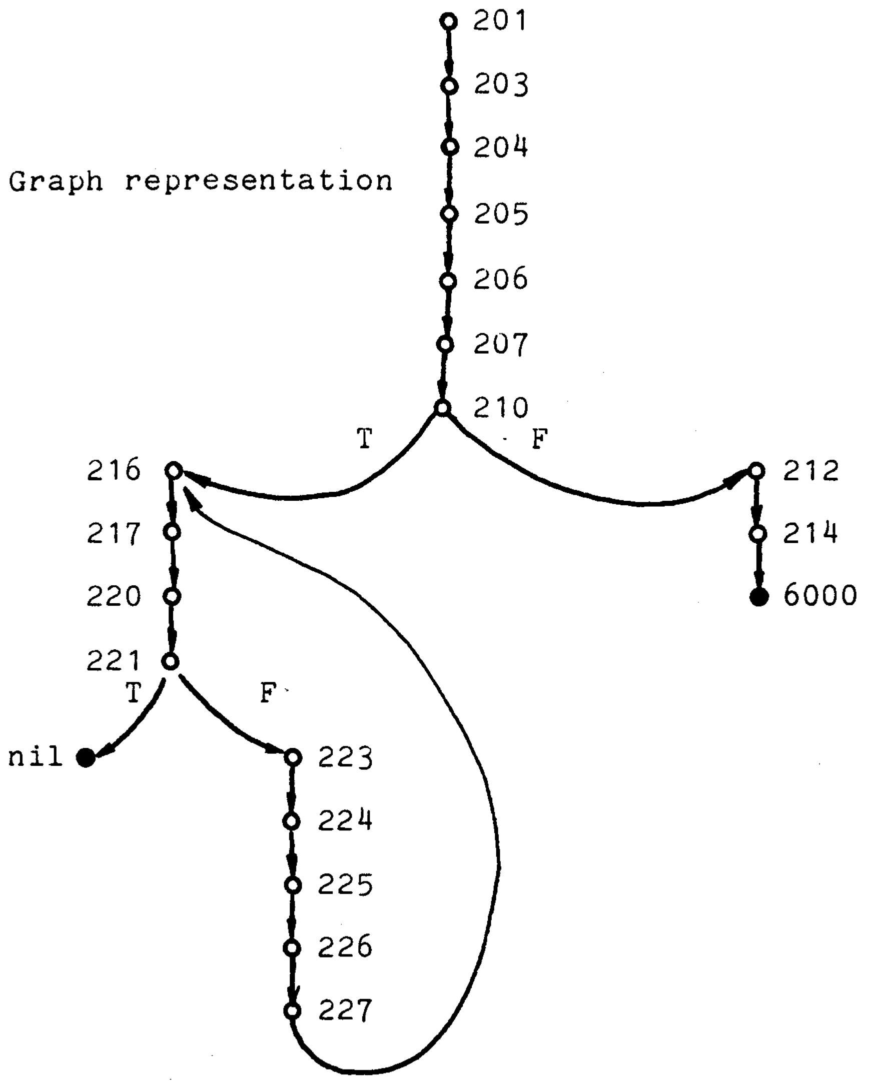


Figure 5-E. Stage one control graph for program of Figure 5-A.

Continuing from 216, we encounter another conditional instruction at 221. This is a conditional RETURN from the ~~subroutine.~~ Since the target of the jump is determined only at execution time, the program trace cannot continue along the "true" path from 221. The "false" path starts at 223 and continues to 227 where we have a loop back to 216. The loop is recorded in the control graph and we back up to trace the "false" path from 210 (having just exhausted the "true" path from that location). Location 214 is a jump to an address outside of the window and is treated as a sink node since the trace cannot continue.

At this point, all of the executable instructions in the window are included in the control graph. The only locations not in the graph are 200, the mark word of the subroutine, and 231-234, the data area. These statements, as well as those which did not generate any code, are handled in the code output section of the decompiler.

Stage Two

The stage one graph has no transfers between sibling subgraphs of a conditional node, so the stage two algorithm does not establish any explicit join nodes as described in Chapter 4. Two nodes (214 and 217) are removed because they are unconditional jump instructions. The control is represented by the edges of the graph making unconditional jump nodes redundant.

```
(201
 203
 204
 205      LISP s-expression representation
 206
 207
 210
 (216 217 220 221 (nil) (223 224 225 226 + * 216))
 (212 + * 6000))
```

Graph representation

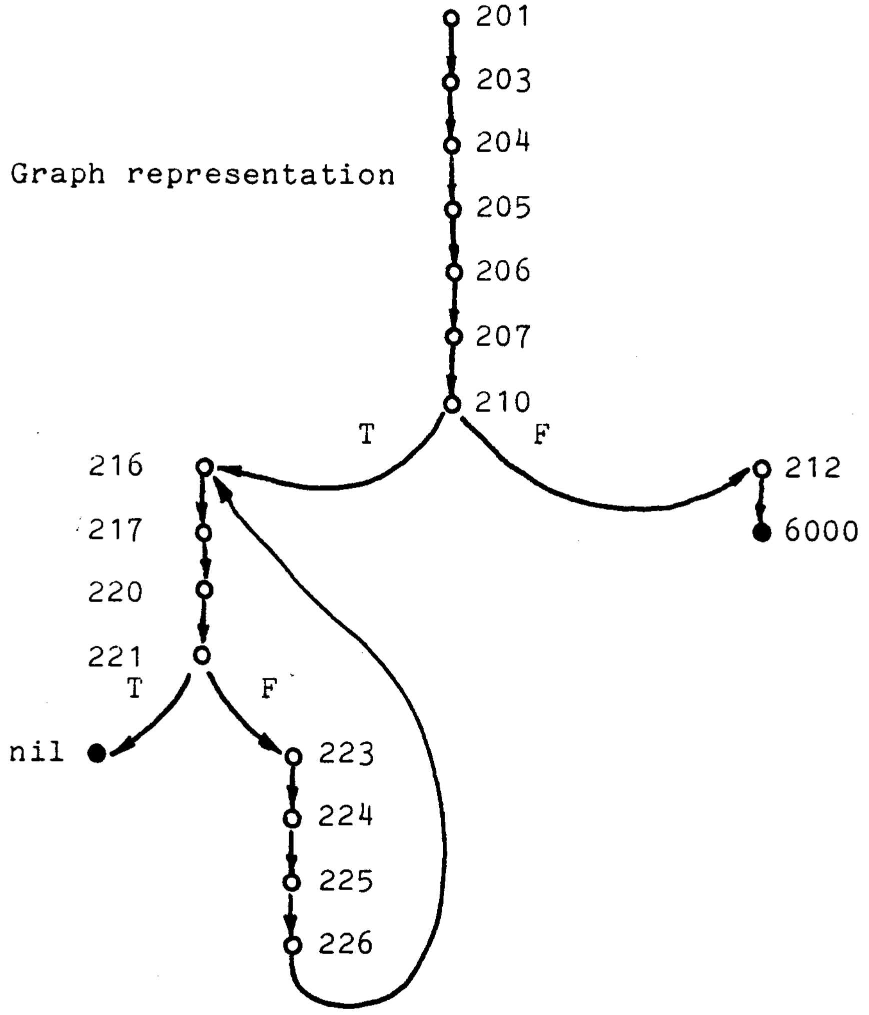


Figure 5-F. Stage two control graph for program of Figure 5-A. (Every join subgraph is empty.)

IMTEXT GENERATION

The stage two control graph of Figure 5-F is passed to the IMTEXT generation routines. The IMTEXT generation routines fill in the nodes of the graph with the corresponding IMTEXT statement(s). The output of the IMTEXT generation routine is shown in Figure 5-G. This section discusses properties of this IMTEXT graph.

The names of registers appear explicitly as AREG!, BREG!, XREG!, OF!. Synthetic variables needed in the IMTEXT statements are represented as !Tn or !Sn (n = 0, 1, 2, ...). Synthetic variables can be considered as temporary result holders.

The instruction at location 207 (and 220) expands into three IMTEXT statements

```
(207 0 MOVE AREG!  !T3)
(207 1 SUB (!T3 =1) !T3)
(207 2 MOVE !T3 AREG!)
```

At first glance, this may seem an over complicated sequence to generate for an instruction which might be represented as

```
(207 0 SUB (AREG! =1) AREG!).
```

The 620/i instruction DAR belongs to a large group of register transfer instructions which can conditionally apply four transformations on the data from three register sources and place the result in three register destinations. The function which translates instructions of this group into IMTEXT does this in a general fashion and does not check for special cases. Of course, special case checks could be added to reduce the number of IMTEXT instructions generated for the common simple cases such as DAR. It is also interesting to note that this group of instructions has 512 (2**9) combinations, but by using the machine code to decipher the instruction rather than trying to recognize all of the possible mnemonics, the IMTEXT generator is greatly simplified. It can use the work the assembler has already done.

Instruction 210 is translated into two parts also. The condition predicate is evaluated and the result is stored in the synthetic variable !T1 by

```
(210 0 LT AREG! !T1).
```

The actual jump instruction is translated as a "branch true," testing the boolean variable !T1:

```
(210 1 BT (!T1 "*+6") NIL).
```

At location 216 we have a LABEL statement introduced because 216 has more than one predecessor (210 and 226).

```
((201 0 CALL (PCHK) (PCHK))
 (203 0 MOVE (P2) AREG!)
 (204 0 ADD (AREG! P3) AREG!)
 (205 0 SUB (AREG! P1) AREG!)
 (206 0 SUB (AREG! H) AREG!)
 (207 0 MOVE (AREG!) !T3)
 (207 1 SUB (!T3 =1) !T3)
 (207 2 MOVE (!T3) AREG!)
 (210 0 LT AREG! !T1)
 (210 1 BT (!T1 "*+6") NIL)
 ((216 0 LABEL (216) NIL)
  (216 1 MOVE (P1) AREG!)
  (217 0 SUB (AREG! P2) AREG!)
  (220 0 MOVE (AREG!) !T3)
  (220 1 SUB (!T3 =1) !T3)
  (220 2 MOVE (!T3) AREG!)
  (221 0 GE AREG! !T1)
  (221 1 INDIRECT (MOVE) !S1)
  (221 2 BT (!T1 !S1) NIL)
  ((NIL NIL NIL NIL NIL))
  ((223 0 INDIRECT (P1) !S1)
   (223 1 MOVE (!S1) AREG!)
   (224 0 INDIRECT (P3) !S1)
   (224 1 MOVE AREG! !S1)
   (225 0 BUMP P1 P1)
   (226 0 BUMP P3 P3)
   (NIL NIL B (216) NIL)))
 ((212 0 MOVE ("=´?H´") AREG!)
  (NIL NIL B (6000) NIL)))
```

Figure 5-G. IMTEXT control graph for sample program derived from stage two control graph of Figure 5-F. (This graph is in the form of a LISP s-expression.)

At location 221 the JAP* instruction illustrates an example of a future forward substitution of a synthetic IMTEXT statement into the argument list of a successor. The target of the jump is indirect through MOVE. The meaning of

(221 1 INDIRECT (MOVE) !S1)

is "evaluate MOVE and store the result in !S1." The second argument of

(221 2 BT (!T1 !S1) NIL)

is the forward substitution target of the INDIRECT statement. The "true" subgraph of 221 is empty since the target of the indirect branch is indeterminant.

At statement 224 we see an example where substitution will occur in the change list of an IMTEXT statement. This corresponds to an expression evaluation on the left of an assignment.

The branch instructions after 212 and 226 in the IMTEXT graph are synthetic. In this case, they correspond to the instructions 214 and 227 although synthetic branch instructions do not have to have a counterpart in the source code.

The IMTEXT statements generated at this step of the decompilation are all <u>basic</u> statements, i.e., they have only simple arguments and change lists. Any nesting of statements is done in the next step -- forward substitution.

FORWARD SUBSTITUTION

The revised IMTEXT control graph after forward substitution is shown in Figure 5-H. Statements 203.0-210.0 compute the value of a predicate which is used to determine whether the last location to be moved by the source program is in the bounds delimited by variable H on the upper limit. The address of the last target location for the move is computed by (P2 + P3 - P1) and then is compared to H by substracting (H + 1). The program is computing the predicate

Is P2 + P3 - P1 <= H ?

The IMTEXT statements use temporary locations AREG! and !T3. The forward substitution algorithms determine that all uses of temporary variables can be suppressed and statements 203.0-210.0 can be condensed into the first argument of 210.1 -- the conditional branch, BT.

It is easy to see that AREG! can be eliminated (statements 212 and 216 limit the downstream scope of the variable use past 210). Built into the forward substitution algorithm is the notion that a synthetic variable (e.g., !T3 in 207) only has scope over its immediate successors in the same subsequence to which it belongs (e.g., 207.0, 207.1, 207.2). The synthetic variable value will never be needed after the subsequence. Using this fact, !T3 can be substituted without explicit assignment at 207.1 and 207.2. Without this convention, we would have to make explicit the assignment to !T3 since we could not determine the busy status of !T3 on exit from MOVE.

The condensation of statements 216.1, 217, 220, and 221.1 into 221.2 is similar to the above example except that the busy status of AREG! is not known due to the indeterminant branch at 221.2. We cannot know whether the calling routine expects a value to be returned in the A-register when MOVE returns.

```
((201 0 CALL (PCHK) (PCHK))
 (210
  1
  BT
  ((210
    0
    LT
    ((207
      2
      MOVE
      ((207
        1
        SUB
        ((207
          0
          MOVE
          ((206
            0
            SUB
            ((205 0 SUB ((204 0 ADD ((203 0 MOVE (P2) NIL) P3) NIL) P1) NIL)
             H)
            NIL))
          NIL)
        =1)
        (!T3)))
      NIL))
    (!T1))
  "#+6")
  (NIL))
 ((216 0 LABEL (216) NIL)
  (221
   2
   BT
   ((221
     0
     GE
     ((NIL
       NIL
       MOVEXP
       ((220
         2
         MOVE
         ((220
           1
           SUB
           ((220 0 MOVE ((217 0 SUB ((216 1 MOVE (P1) NIL) P2) NIL)) NIL) =1)
           (!T3)))
         NIL))
       (AREG!)))
     (!T1 AREG!))
    (221 1 INDIRECT (MOVE) (!S1)))
   (NIL AREG!))
  ((NIL NIL NIL NIL NIL))
  ((224
    1
    MOVE
    ((NIL NIL
          MOVEXP
          ((223 1 MOVE ((223 0 INDIRECT (P1) (!S1))) NIL))
          (AREG!)))
    ((224 0 INDIRECT (P3) !S1) AREG!))
   (225 0 BUMP P1 P1)
   (226 0 BUMP P3 P3)
   (NIL NIL B (216) NIL)))
 ((212 0 MOVE ("=´?H´") AREG!) (NIL NIL B (6000) NIL)))
```

Figure 5-H. IMTEXT control graph in LISP s-expression form after forward substitution was applied to the graph of Figure 5-G.

This is an example of helpful information which is not available. (See Chapter 3 for the discussion of this topic.) If such information was included in the attribute list of MOVE, e.g., "MOVE returns no information to the calling routine in the A-register; The A-register is destroyed," then we would know the explicit assignment to AREG! at 220.2 is not needed. In fact, we do not know this so we assume the worst case.

At statement 223.1 the same situation occurs. Statement 221.2 is downstream of 223.1 (in the loop) so we make the store to AREG! explicit (MOVEXP).

Statement 224.0 has been substituted in the change list of 224.1 for the synthetic variable !S1.

Once the forward substitution has been completed, the decompiler begins to generate the target code.

TARGET CODE GENERATION

The generation of the target code proceeds as follows. The root of the IMTEXT control graph is at location 201. Statements in the source program prior to line 14 (Figure 5-A), the line which generated the root node, are examined by the pseudo-op processor and appropriate code is generated. The output corresponds to lines 1-14 in Figure 5-I.

The structure recognition algorithm then attempts to match structure patterns with the control graph as discussed in Chapter 4. In the case of this sample program, the structures recognized are the IF statement conditional pattern and the pre-test WHILE loop pattern. The loop is contained in the body of the conditional. An alternative and equivalent pattern would have the loop outside the body of the conditional and the error exit code inside. This alternate structure might be favored over the one actually selected, since it reduces the nesting level of a large volume of code (the loop) and seems to make the error exit more visible. If such stylistic conventions can be codified, they can be added to the code generators to cause certain equivalent constructs to be favored over others.

An example of such a stylistic transformation programmed into the decompiler is one which, under appropriate conditions, raises the level of an ELSE clause in an IF statement if the last statement of the THEN clause is an unconditional branch, i.e.,

```
     IF exp THEN BEGIN S1; ...  GOTO L; END;
          ELSE S2;
```

becomes

```
     IF exp THEN BEGIN S1; ...  GOTO L; END;
     S2;.
```

Similar transformations can be applied to argument expressions. The output of the target code for the predicate of the BT statement at 210 in the IMTEXT control graph uses an "identity" discussed earlier in this chapter, namely, for variables A and B

$$A - B - 1 < 0$$

becomes

$$A - B <= 0$$

and finally,

$$A <= B.$$

The application of this transformation succeeds in changing the expression

$$P2 + P3 - P1 - H - 1 < 0$$

into

$$P2 + P3 - P1 <= H$$

(line 16 of Figure 5-I), a considerable improvement in readability.

The decompiler was not as successful in rearranging the IMTEXT expression of statement 221 (Figure 5-H) to yield line 22 of the target program (Figure 5-I). The explicit assignment in

$$(AREG! := P1 - P2 -1) < 0$$

defeats application of the transformation. We discussed earlier that this explicit assignment was not really necessary to the program although the decompiler did not determine that fact due to the lack of information. The predicate could be rewritten as

$$P1 <= P2$$

if the assignment were removed. This example illustrates how lack of information at one stage of the decompilation can affect later stages due to "worst case" assumptions. In this case, lack of helpful information causes stylistic downgrading of the target program.

After the decompiler outputs the code for the procedure itself, it calls the pseudo-op processor to output the data declarations (target lines 40-45 of Figure 5-I) and the decompiler writes the target program to a file to be read by the post-processor.

POSTPROCESSOR

The input to the postprocessor contains many symbols not in the target language (e.g., ´<<´, ´/]´, AREG!), as well as the transliterated symbols introduced by the preprocessor. One of the tasks of the postprocessor is to translate these symbols into their proper representations.

Another important task of the postprocessor is to format the target program according to some rules about indentation, comment placement, and spacing which will make the output more readable. Of course, these rules are subjective for a free format language. As much or as little effort can be expended in the formatting procedure as is desired. The meaning of the program, in terms of its executability, is unaffected by this process, but the value of a well formatted presentation of the program should not be underestimated with respect to its understandability.

Figure 5-J shows the output of the postprocessor for the sample program. No manual changes were needed.

```
 1(("%*%")
 2 ("%*%")
 3 ("%* SAMPLE PROGRAM TO DECOMPILE%")
 4 ("%*%")
 5 (^ /; ORG /, (200) ^ ; "%FIRST ADDRESS OF PGM%")
 6 (EQU PCHK = (5000) ; "%EXTERNAL SUBROUTINE%")
 7 (EQU ERR = (6000) ; "%EXTERNAL ERROR ROUTINE%")
 8 ("%*%")
 9 ("%*MOVE - M <START\056SOURCE> <END\056SOURCE>
                                      <START\056TARGET>
10 ("%* MOVE WORDS [P1,P2] TO [P3,P3+P2-P1]%")
11 ("%* MUST HAVE P1<=P2, P2<=H, L<=P1, P3+P2-P1<=H%")
12 ("%* ELSE GOTO TO ERROR ROUTINE%")
13 ("%*%")
14 (PROCEDURE MOVE ; % ADDRESS 200 % "%ENTRY POINT%")
15 ((CALL PCHK ; "%CHECK PARMS P1,P2%")
16  (IF ((<< ((<< (P2 + P3) >>) - P1) >>) <= H)
17      "%\050FINAL TARGET\051 > H\073 ERROR\056%"
18      THEN
19      BEGIN
20      ((L001 : "%LOOP HEAD%")
21       (WHILE
22        ((<< (AREG! := ((<< (P1 - P2) >>) - 1)) >>) < 0)
23        "%CHECK TERMINATION%"
24        "%DONE%"
25        DO
26        BEGIN
27        (((/[ P3 /]) :=
28                      (AREG! := (/[ P1 /]))
29                      ;
30                      "%GET SOURCE WORD%"
31                      "%STORE IT%")
32         (BUMP P1 ; "%BUMP SOURCE PTR%")
33         (BUMP P3 ;))
34        END;)
35       (RETURN ;))
36      END;)
37  (AREG! := "=´?H´" ; "%ERROR CODE%")
38  (GOTO ERR ;))
39 ENDP;)
40(("%*%")
41 (DECLARE P1 = (0) ;)
42 (DECLARE P2 = (0) ;)
43 (DECLARE P3 = (0) ;)
44 (DECLARE H = (0) ;)
45 (DONE \056))
```

Figure 5-I. Output of decompiler code generation routines before postprocessing (line numbers added for reference).

```
% SAMPLE PROGRAM TO DECOMPILE %

",ORG,0200" ;                 % FIRST ADDRESS OF PGM %
EQU PCHK = 05000 ;            % EXTERNAL SUBROUTINE %
EQU ERR = 06000 ;             % EXTERNAL ERROR ROUTINE %

% MOVE - M <START.SOURCE> <END.SOURCE> <START.TARGET> %
% MOVE WORDS [P1,P2] TO [P3,P3+P2-P1] %
% MUST HAVE P1<=P2, P2<=H, L<=P1, P3+P2-P1<=H %
% ELSE GOTO TO ERROR ROUTINE %

PROCEDURE MOVE ;              % ADDRESS 200. ENTRY POINT %
        CALL PCHK ;           % CHECK PARMS P1,P2 %
        IF ((P2 + P3) - P1) <= H
                              % (FINAL TARGET) > H; ERROR. %
        THEN
          BEGIN
L001 :                        % LOOP HEAD %
            WHILE (AREG := (P1 - P2) - 1) < 0
                              % CHECK TERMINATION %
                              % DONE %
            DO
              BEGIN
                [P3] := AREG := [P1] ;
                              % GET SOURCE WORD %
                              % STORE IT %
                BUMP P1 ;     % BUMP SOURCE PTR %
                BUMP P3 ;
              END;
            RETURN ;
          END;
        AREG := ´?H´ ;        % ERROR CODE %
        GOTO ERR ;
ENDP;

DECLARE P1 = 0 ;
DECLARE P2 = 0 ;
DECLARE P3 = 0 ;
DECLARE H = 0 ;
DONE.
```

Figure 5-J. Output of postprocessor of decompiler.

SUMMARY

In this chapter we have presented some details of the decompiler implementation and taken a small sample program through each step of the decompilation process to its final representation in the target language. A comparison of the source program (Figure 5-A) with the target program (Figure 5-J) makes clear what has been done. A linear sequence of instructions has been transformed into an equivalent representation in the target language. The scope of each conditional and looping construct is clearly identifiable. The computations involved in the conditional predicates have been collected into expressions associated with the instructions groups they control.

This example is quite simple and easy to understand in its assembler language form. The target program contains some extraneous information and might be rearranged slightly to suit different tastes. However, most programmers would probably agree that the target language program is more readable and understandable than the original.

Chapter 6

DECOMPILATION EXPERIMENTS

INTRODUCTION

In the last two chapters we discussed the algorithms and details of the decompiler implemented during this research. We illustrated the action of the decompiler on a small sample program. In this chapter we present the results of decompiling two larger source programs. The goal of these experiments is to provide realistic information regarding the following questions:

- How much manual intervention is necessary, and what kind?
- What is the relationship of the machine code volume of the source program to the machine code volume of the recompiled target program?
- What can we say about the speed of the decompiler itself?
- What is the relationship of language constructs of the source program to those of the target program text generated by the decompiler?
- How do the experimental results presented here compare with those of other decompilation efforts?

THE TEST PLAN

The test plan consists of using the decompiler to produce four different representations of the test program. In addition to the initial source version, P0, we have

P1--produced by the postprocessor of the decompiler.

P2--Manual changes are made to P1 which make the resulting program compilable by MOL620 and cause it to execute correctly after recompilation.

P3--Data movement which is "obviously redundant" is removed from P2. This phase approximates what a decompiler with more helpful information might produce.

P4--The P3 version is rewritten maintaining functional equivalence. This represents the results of applying an "ultimate" decompiler to produce the target program. Information which realistically could only be expected to be gleaned by a human is used to rearrange or rewrite entire functions or groups of functions in a more readable and more organized fashion.

P4 represents the best program we could hope a decompiler to produce and thus serves as a limit point of our expectations. The difference in the object code sizes of P0 and the recompiled version of P4 should be attributable to the implementation language (barring some gross deviation from the "best" algorithms possible which could be expressed in the implementation language).

CHOOSING A TEST CASE

The attributes of a source program which would be a satisfactory test case for the decompiler are

- o relatively <u>large</u> -- over one thousand source statements. This would insure that many different code sequences would probably appear in the source program and cause most of the parts of the decompiler to be exercised.
- o <u>average quality</u> -- representative of code that might be produced by an "average" programmer, i.e., and not too full of tricks and not too well structured.
- o <u>mature</u> -- in use for a long period of time in a production environment; patches and improvements have been made by several different people over a period of time.
- o <u>useful</u> -- the task performed by the program is currently useful to some community of users.
- o <u>testable</u> -- proper operation of the target program can be verified by executing it. It does not use equipment (e.g., I/O equipment) unavailable to the tester.
- o <u>familiar</u> -- knowledge of the program´s structure and performance would be helpful in assessing the

results of the decompilation.

- <u>candidate for evolutionary improvements</u> -- a successful decompilation should make such improvements easier to implement in the target language.

THE FIRST TEST CASE -- ISADORA

The first large program selected for decompilation was an interactive debugger for the Varian 620/i computer. This program, called ISADORA, was written in assembler language in 1969 by an undergraduate student in the UCI Department of Information and Computer Science.

ISADORA contains 2041 source lines consisting of

523	comment lines,
49	listing control lines (e.g., EJECT),
70	non-code producing lines (e.g., EQU, ORG),
1199	machine instructions, and
200	data declarations (code producing).

The production version of ISADORA occupies 2268 16-bit words of memory.

The coding style in ISADORA is typical of many programs written by novice assembler language programmers. There are some attempts to save code by tricks or non-obvious code sequences. The program is more modular than most in that there are many subroutines but entry, exit and variable sharing among them is often undisciplined. Some code modification is done, particularly in in-line argument lists. The nature of the programming environment on which ISADORA was originally prepared influenced some of these coding techniques. A slow, tedious paper tape I/O system with teletype hard copy output contributed to the formation of small code packets which could be edited (manually) and assembled separately. Lack of symbol table space in the original 8K machine led to the use of many "*+n" (location counter relative) constructs instead of labels. For example, "JMP *+19" was not an uncommon type of instruction to appear in the program.

ISADORA, in use for six years on the departmental minicomputer, had been modified or extended many times, usually in small increments, by at least four students in addition to the original author. The occurrences of relative addressing mentioned above became a source of maintenance problems and most of them were removed over the years. (The memory on the 620/i system had been expanded to 32K and symbol table space was no longer a problem when ISADORA was assembled.) Some of these "*+n" addressing constructs still remained when the program was decompiled, however.

ISADORA is the best debugger available to users of the 620/i and so is used often. The nature of the program made it easy to verify correct operation. The inclusion of the 620/i in the Distributed Computing System (DCS) research network (Farber et.al. 1973) in 1972 meant the computer was being used and programmed much more heavily than in the past. Enhancements of a major nature to ISADORA were not attempted because no one wished to undertake modification to the assembler language program. A decompilation of ISADORA, if successful, would translate the text into the system implementation language for the 620/i (developed after ISADORA had already been written) and modification and major improvements would be easier to make. (The DCS system is programmed almost entirely in MOL620 (Hopwood 1971) and MOLSUE (Hopwood 1976), two machine oriented higher level languages.)

With the above qualifications, ISADORA was chosen as a good candidate for the initial large scale test of the decompiler.

ISADORA THROUGH THE DECOMPILER: P0 -> P1

The ISADORA source program (P0) was assembled on the PDP-10 cross-assembler. (See Appendix IV for an excerpt of the ISADORA program input to the decompiler.) The output listing was passed through the preprocessor of the decompiler to yield the formatted code and the symbol table. This information passed through the LISP portion of the decompiler as discussed in earlier chapters. Necessary information regarding the nature of subroutine calls was requested by the decompiler and entered by the user. (See chapter 5 for an example.) Progress of the decompilation was monitored on the user's terminal. The output of the code generation phase was then passed to the postprocessor yielding P1.

Up to this point the information supplied to the decompiler by the user was the assembler output listing, a value for the maximum window size desired, necessary information regarding subroutines, and names of input/output files. Only six subroutines had attributes not covered by the default attributes assumed by the decompiler.

ISADORA MANUAL CHANGES: P1 -> P2

The output of the postprocessor (P1) may be suitable for the purposes of understanding or documentation but if we wish to retranslate the target program back to machine language and execute it, we must perform some manual editing. In other words, the output of the decompiler is not directly acceptable to the MOL620 compiler.

The manual changes needed to transform the output of the decompiler into a compilable and executable program can be classified into five groups:

1. communication with an undecompiled program,
2. interrupt service routines,
3. self-modifying code,
4. symbol definitions, and
5. unimplemented instructions.

The manual changes needed to change the output of the postprocessor (P1) into a compilable, executable, target program are summarized in Table 6-A.

Table 6-A. Summary of manual changes made to P1 (ISADORA).

Environment transitions	
subroutine calls	2
interrupt service routines	14
Self-modifying code	
computed jump addresses	4
temporary values	7
I/O instructions	12
Symbols	
duplicates	2
wrong place	5
new symbol definitions	1
equivalent symbols	3
entry points	4
Unimplemented code	
execute instruction	1
Total number of changes	55
Number of hours required	9

A manual change is considered to be a change to a string of characters, usually adding, deleting, or altering some symbol in the target program. Some of the changes to symbols were accomplished with one text editor command such as "substitute for all occurrences..." operating on the entire file. The important factor in interpreting the manual changes data is not the number of them but rather the time required to produce the changes. The total time to make the manual changes and verify that the result executed correctly was about nine hours. (See Appendix V for a an excerpt of the P2 version of the ISADORA source program.)

We now discuss each group of manual changes needed to cause the P1 version of ISADORA to execute correctly.

<u>Communication</u>

We define the <u>memory environment</u> of a program to be the set of all memory elements accessible by the program. Let the environment of an executing decompiled program be the decompiled environment. The environment of an executing program which has not been decompiled is the external environment. We assume the set of primitive operations directed by each program are the same, i.e., they are both written in the machine language of the same machine.

In order for two programs to communicate with each other, their memory environments must overlap in a least one place, e.g., a register, core location, I/O interface, or satellite relay station. A <u>communications protocol</u> establishes what parts of the overlapping environment are used to transmit information, what time the information is transmitted, and the nature of the contents of the shared elements.

When a program is decompiled, its memory environment is artificially changed so that operations to the physical registers of its source machine are now performed upon pseudo registers (e.g., the memory variables AREG, BREG, XREG). If the physical registers served as a communication path with other programs whose environment has not been altered in conformity with the decompiled program, then the communications protocol is destroyed. The pseudo registers of the decompiled program no longer overlap with the registers of the undecompiled program with which it wishes to communicate.

Figure 6-Aa illustrates the communication memory environment overlap of programs D and P which communicate through one memory element R. Figure 6-Ab shows the decompiled version of D, D′, now attempting to communicate with P through pseudo register R′, but R′ is not in the communication memory enviroment of P. P still attempts to communicate with D′ through R. The simple result of this confusion is that P and D (D′) no longer communicate correctly because their communications protocol is no longer valid. Attempts to execute these programs together will result in errors. Figure 6-Ac show the introduction of a conduit process, C, which moves information between R and R′ at the appropriate times thus completing the path between the two memory environments.

ISADORA communicates with two external routines in the 620/i system which were not decompiled. A sequence of code implementing a conduit process had to be placed at each call site to the external routines to transfer subroutine arguments from the pseudo registers to the machine registers.

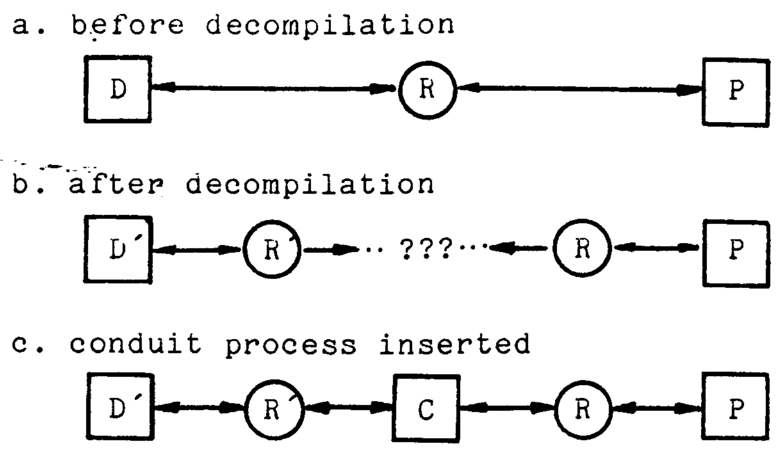


Figure 6-A a,b,c. Communication between decompiled (D) and undecompiled environments (P) through register R.

Interrupts

We have seen that overlap of memory environments can cause problems when decompiled programs try to communicate with ones which were not decompiled. This problem has another aspect which was revealed the first time the decompiled ISADORA was executed.

Program memory environments may overlap for reasons other than communications. Machine registers and status words are almost always shared between different programs operating in the same computer in a multiprogrammed environment. If these programs belong to different execution environments this overlapped memory must be saved and restored when execution environments are switched or the programs will interfere with each other.

Figure 6-B shows a diagram indicating two execution environments, D and P. The arrows indicate possible transitions among various programs in these environments. (The numbers on the arrows are labels.) For example, assume interrupts can cause the transitions indicated.

$$\begin{tikzcd}
\boxed{D} \arrow[loop left, "1"] \arrow[r, bend right, "3"'] & \boxed{P} \arrow[loop right, "4"] \arrow[l, bend right, "2"']
\end{tikzcd}$$

Figure 6-B. Environment transitions due to interrupts.

Let the environment D correspond to the execution environment of ISADORA. Let the environment P correspond to the execution environment of any programs not decompiled but sharing processing time and some memory elements with ISADORA. Such a system exhibits the transitions of Figure 6-B when external interrupts are allowed. An interrupt service routine is executed, and control is returned to the point of interruption with its context restored.

Transitions 1 and 2 are of interest here. How did the decompilation of ISADORA affect the save/restore code in the interrupt service routines of ISADORA? We can answer that question by examining the memory environment of the interrupt service routines and of the code which is interruptable. Overlap of these memory environments implies that the contents of the overlapped elements must be saved and restored by the interrupt service routine.

Transition 1 interrupts are from the decompiled environment back into the decompiled environment. The memory environment overlap in the interrupt service routine is the set of physical registers <u>and</u> the pseudo registers. The undecompiled interrupt service routine saved and later restored the physical registers. The decompiled service routine saves and restores only the pseudo registers. The physical registers are <u>not</u> saved and restored although they need to be. This must be done to avoid destroying the context of the interrupted routine. The practical resolution of this problem is to insert code sequences to save and restore the physical registers into all interrupt service routines handling type 1 environment transitions. The decompiled code which saves the pseudo registers is also still required.

Transition 2 interrupts are from the external environment into the decompiled environment. The shared memory environment consists of the physical registers. The situation is similar to that described earlier for subroutines communicating between the external and decompiled environment. On entry to the decompiled environment, the physical registers must be copied to the pseudo registers. On return to the external environment the pseudo registers must be copied back to the physical registers. (This is the action of the conduit process.) The decompiled routine will save and restore the pseudo registers to maintain the interrupted context.

Transition 3 interrupts from the decompiled environment to the external environment do not cause any need for changes to the programs involved. Only shared memory must be saved and restored and decompilation has not changed the set of shared memory locations between the decompiled code and external interrupt service routines. Transition 4 interrupts are not affected by the decompilation process and so they are handled as before.

Figure 6-C shows two simple interrupt routines before and after decompilation. The first handles transition 1 interrupts. The physical A-register is saved and restored from the variable $A. The second handles transition 2 interrupts. The physical A-register is transferred to the pseudo register AREG. (Assume for simplicity that only the accumulator is in the shared memory environment.)

The manual intervention needed to preserve or copy information between the decompiled program and external programs is somewhat tedious if there are many interrupt service routines or calls to external routines. The information about which routines are external and which are interrupt service routines could be added to the subroutine attribute information given the decompiler so that appropriate code could be inserted automatically during decompilation. The time needed to make manual changes for environment transitions accounted for about one-half of the total time expended making all of the changes to ISADORA.

```
Before Decompilation                 After Decompilation

                      Transition 1
ENTRY:                          ENTRY:
                                     $A := (AR) ; %ADDED%
       STA SAVEA                     SAVEA := AREG ;
       <PROCESS INTRPT>              <PROCESS INTRPT>
       LDA SAVEA                     AREG := SAVEA ;
                                     (AR) := $A ; %ADDED%
       RETN                          RETURN;
                                     DECLARE $A ;

                      Transition 2
ENTRY:                          ENTRY:
                                     AREG := (AR) ; %ADDED%
       STA SAVEA                     SAVEA := AREG ;
       <PROCESS INTRPT>              <PROCESS INTRPT>
       LDA SAVEA                     AREG := SAVEA ;
                                     (AR) := AREG ; %ADDED%
       RETN                          RETURN ;
```

Figure 6-C. Changes needed in interrupt service routines after decompilation.

Self-Modifying Code

Self-modifying code occurs in many assembler language programs for two reasons -- to save space or to improve execution speed. The decompiler developed in this research recognizes self-modifying code by marking as "instruction" memory locations which are traced in the control graph generation routines. At IMTEXT generation time when individual instructions are interpreted to determine their meaning, those locations which would be modified by the instruction are marked "writable." After processing the window information and generating the target code, locations which are marked "writable" and "instruction" are potential targets of instruction modifications.

As implemented, this technique has two deficiencies:

1. Only information about memory locations in the current window is kept. This means that instructions in the window which modify code outside of the window are not detected.
2. Modification of code through indirection or indexing, where the target of the modification is unknown at decompilation time, is not detected.

The first problem can be solved by having the decompiler keep information about the source program that is not in the window (as is done with subroutine attributes). This increases the memory requirements of the decompiler but is feasible. The second problem is another variation of the difficulty of indeterminant addressing. Short of executing (symbolically or actually) the program on its set of valid input data we cannot hope to overcome this problem in general.

In ISADORA, self-modifying code appears for three reasons:

1. computing branch addresses in the second word of a JUMP instruction,
2. using the second word of a double-word instruction as a temporary variable, and
3. modifying I/O instructions to change device addresses.

In machines like the 620/i which do not have shift instructions allowing variable shift counts, it is common to modify the count field of the shift instruction; however, no occurrences of this type of self-modification appeared in ISADORA.

The first two cases mentioned above were recognized by the decompiler and announced in the target program text (P1) with a warning message. The third case was not. The modifications to the I/O instructions were done by a single subroutine and all of the instructions modified were in other routines outside of the window. It took only a few minutes using an interactive text editor to eliminate this routine and develop a more general equivalent scheme using the built-in I/O functions and function calls generated for the I/O instructions by the decompiler.

The important point to emphasize in this discussion is that self-modifying code which cannot be readily translated by the decompiler does not pose severe problems for human beings to solve. In most programs written after the introduction of machines with index registers, self-modifying code appears very infrequently relative to the entire code volume. It is easy to recognize this code in the target program text. Manual changes to the code do not take a significant amount of time to accomplish. Most occurrences are flagged by the decompiler procedure mentioned above. Those which are not detected by the decompiler are easily recognized by reading the target code itself.

Symbols

Manual changes to the decompiled target program were needed to change, add, or remove symbolic names from the text. There are five reasons why changes to symbols had to be made in ISADORA.

First, duplicates were removed. Under certain conditions (when creating synthetic procedures) the decompiler would output a label definition and then create a procedure heading using the same symbol. For example,

```
NAME: PROCEDURE NAME;
or
PROCEDURE NAME; NAME:
```

The label occurrence NAME should be deleted.

Second, labels sometimes appear in syntactically incorrect positions and must be moved. For example, a label may not appear in the following context in the target language

```
IF exp THEN label: BEGIN ... END.
```

We move the label so that the statement is syntactically correct, as in

```
IF exp THEN BEGIN label: ... END.
```

Third, symbol definitions were inserted at the beginning of the program. In the source program, memory locations may be referred to by actual numeric memory address rather than by a symbolic name. The target language does not permit this. The decompiler generates names of the form "Lnnn" to use in place of the numeric addresses. These definitions need to be put in the target program. The user can choose more appropriate mnemonic names to replace the "Lnnn" symbols if he wishes.

Fourth, the uses of equivalent symbols are changed in certain contexts. When the decompiler looks up a numeric address in the symbol table, there may be several names, chosen by the programmer for mnemonic content, that have the same numeric (assembly time) value. The decompiler chooses the first one it finds. This may not be the symbol the programmer wishes to use. The desired symbolic substitutions can be made with a text editor. (This process is not actually necessary for proper compilation or execution of the target program as long as the equivalent symbols are not later redefined so they are no longer equivalent.)

Fifth, entry points are added to procedures. Entry points to code in the window from code outside the window are not labelled in the target program. The analysis of the code in the window does not reveal entry points of this type. Labels of statements in the source code without multiple predecessors <u>in the window</u> are discarded as not necessary since a join in the graph is not indicated. The first recompilation of the target program will show these entry points as undefined. The decompiler could copy all labels in the source program into the target program but the use of entry points into the middle of subroutines from some external routine is bad programming practice. The code should probably be modified to eliminate these entries.

The manual changes mentioned above could be eliminated by altering the decompiler or the target language compiler. However, the additional effort needed to accomplish this editing is small. The added complexity to the decompiler is probably not worth the savings anticipated except perhaps in a high volume production system.

Unimplemented Operations

The decompiler does not attempt to translate the Execute instruction of the 620/i into a compilable sequence. When it occurs in a source program it is indicated in the target program and is left for the user to rewrite. The target language does not support the Execute instruction. It occurs infrequently in 620/i programs (once in ISADORA). Manual translation to an alternate statement is very easy.

SIMPLE OPTIMIZING OF ISADORA: P2 -> P3

After making the manual changes necessary to compile and execute the output of the decompiler, the next phase of the experiment involved taking P2, the executing decompiled ISADORA, and manually applying some simple transformations to the code approximating the operation of a decompiler that had more information about the variable usage in the subroutines of ISADORA.

The majority of the changes involved removal of unnecessary assignments to the A-register. These assignments appeared because the variable usage analysis routines did not have enough information to determine that the assignment could be eliminated. For example, in the statements

```
VAR := AREG := Z ; CALL SUBR ;
```

might be changed to

```
VAR := Z ; CALL SUBR ;
```

because AREG is not needed on entry to SUBR and is changed inside of SUBR. The decompiler did not have information about the use of AREG in SUBR so assumed that the value of the AREG was needed.

A similar situation occurs on exit from a subroutine. For example, in the statements

```
VAR := AREG := Z ; RETURN ;
```

the assignment to AREG can be eliminated if the value of AREG after exit from the subroutine is not used.

Making these changes to the target program requires that the variable usage of the subroutines be understood. Removal of a necessary assignment would cause incorrect execution of the program. Table 6-B summarizes the number and types of changes to P2 which produced P3. These changes required approximately four hours to accomplish and verify correct execution of the resulting program.

As might be expected for a single accumulator machine such as ~~the~~ 620/i, the number of changes removing redundant loads to the accumulator account for over half of the total number. On the other hand, less than 15% of the number of changes accounted for about one half of the total savings in program size. These changes dealt with the code which manipulated or tested the overflow indicator. In ISADORA this indicator is used as a one-bit flag rather than as simply arithmetic status -- a somewhat risky programming practice. The decompilation of overflow status manipulation is particularly inefficient. This is due to the fact that the overflow indicator has special instructions to set, reset, and test it. When the hardware bit is translated into a pseudo variable, normal memory accessing instructions must be used which are not as compact. The jump-on-overflow instruction also causes the overflow status to be reset. This fact is faithfully represented in the target code by several instructions, but ISADORA does not depend on this side effect of overflow testing. As a result, the decompiler generated instructions had no real purpose in the target program and were removed by hand during the optimization of P2.

Table 6-B. Summary of simple optimizations made to ISADORA, P2 -> P3.

Remove redundant loads of AREG	69
Remove redundant loads of BREG	6
Remove redundant loads of XREG	8
BREG:=BREG+1 becomes BUMP BREG	1
XREG:=XREG+1 becomes BUMP XREG	1
+1+1 becomes +2	1
Rework overflow testing	14*
Miscellaneous	5
Total number of changes	105
Number of hours	4
Number of words saved (out of 3161)	278

* The overflow testing changes saved an average of 10 words per occurrence.

EXTENSIVE REWRITE OF ISADORA: P3 -> P4

The final step in the experiment with ISADORA involved a manual rewrite of the program (P3) produced by simple manual optimization described earlier. The goal of this process was to produce a program which was better organized, more easily modified, more compact, and a better candidate for contemplated future improvements. The primary goal was to improve the logical organization of the program (or structure) which suffered from several deficiencies:

1. Global variables were used by many different routines in a confusing manner, particularly the

overflow indicator which was used as a flag. These uses were eliminated or standardized.

2. There were many different ways subroutine arguments were passed and values returned. These were standardized using MOL620 argument passing conventions, i.e., arguments were passed and returned in the physical registers.

3. Subroutine entry and exit were undisciplined. Entry was made into the middle of subroutines. Many subroutines did not return at all but branched off somewhere else thereby prohibiting multiple uses of the routine. All routines except fatal error handlers were recoded to return to the caller.

In general, the organization and algorithms used in the original ISADORA were retained or modified only in small ways, but there were two important sections of ISADORA which exhibited all of the deficiencies mentioned above. These were the expression evaluation routines and the routines which implemented the step and breakpoint features. The step feature allowed a user to single-cycle the program under test, i.e., execute one instruction at a time. It did this by <u>interpreting</u> the instruction to be executed and then returning control back to the user's terminal. The expression evaluation routines read arithmetic expressions from the user's terminal, evaluated them, converted types, and performed the indicated arithmetic operations. Although both of these portions of ISADORA executed correctly after decompiling them, they were considered to be too poorly coded to provide a firm foundation for future improvements.

The expression evaluation routines were rewritten. No additional capability was provided but the organization of the code was improved. The step and breakpoint features were completely rewritten since the original version was quite hopeless. The new structure of these functions allowed the introduction of a new and very valuable capability -- continuous interpretation. The entire step function was made into a proper subroutine. Interpretation consists of repeated execution of the step subroutine. This new organization, combined with checks for instruction and address validity (not present in the original) allows interpretive execution of the program being debugged while protecting the external environment from being destroyed by errant jumps or stores. This interpretive feature was introduced at very low cost in terms of programming time and program size after the step function had been reorganized.

The code of the breakpoint facility was rewritten and became more flexible and easier to understand, although from an ISADORA user´s viewpoint the only outward change was an increase in the number of breakpoints which could be set in his program. The number of breakpoints had previously been programmed as constants into ISADORA in several different subroutines. This fact had discouraged earlier attempts at increasing the number of breakpoints.

This reprogramming effort affected nearly every source line of ISADORA in some fashion and took about forty hours to complete and verify correct execution. (Later, as a result of the increased readability and improved structure of ISADORA, two more important improvements were made -- a trace feature was added which keeps track of the jump history of an interpreted program, along with a feature which allows memory access keys (read, write, and execute) to be set on intervals of memory addresses. The important point to make regarding this part of the experiment is that because of the decompilation of ISADORA from assembler language into MOL620, the door was opened to a new round of evolutionary improvement that had not been attempted in the past.

ISADORA CODE SIZE

Table 6-C summarizes the effect the various stages of reprogramming had upon the size of ISADORA in terms of the number of 16-bit machine words needed to represent the program on the 620/i. The number of data words (non-executable) remained unchanged for the P1, P2, and P3 versions. It increased slightly for the extensive rewrite, P4.

Table 6-C. ISADORA machine code size summary.

ISADORA	# words(a)		expansion factor(b)		hours
Version	min	max	min	max	to create
P0 (c)	2133	2268	1.00	1.00	--
(d)	1823	1958	1.00	1.00	
P1	----- not compiled or executed -----				
P2	3501	4294	1.64	1.89	9
	3161	3954	1.73	2.02	
P3	3223	3882	1.51	1.71	4
	2883	3542	1.58	1.81	
P4	2664	3004	1.25	1.32	39
	2301	2641	1.26	1.35	

Notes

a. min is minimum number of words in program using low core pointers and literal pool.

max is maximum number of words in program using double word instructions instead of single word instructions with low core pointers or literals.

b. min expansion factor for program Pn is calculated as

$$\frac{\text{min of version Pn}}{\text{min of version P0}}$$

max expansion factor for program Pn is calculated as

$$\frac{\text{max of version Pn}}{\text{max of version P0}}$$

c. The first line of information refers to program size including any data areas.

d. The second line refers to the program size without data area, i.e., executable code only.

THE SECOND TEST CASE -- TECO

The second test program chosen for decompilation was TECO, an interactive text editor for the 620/i. This program was originally written at the University of Oregon (Eugene) and has been updated by several students at UCI since 1972. The editor is very similar to TECO on the DECsystem-10 including full macro capability.

TECO contains 3211 source lines consisting of

629	comment lines,
64	listing control lines (e.g., EJECT),
142	non-code producing lines (e.g., EQU, ORG),
1998	machine instructions, and
378	data declarations (code producing).

The production version of TECO occupies 4007 16-bit words of memory, excluding its 8K text buffer.

TECO was translated from its initial assembler language version, P0, using a test plan similar to that for ISADORA. P1 was the output of the decompiler; P2 was the result of manual changes made to P1 to cause it to compile and execute correctly. P3 was the result of a minor manual optimization pass. P4 translation was omitted. This was the extensive rewrite phase in the case of ISADORA.

TECO THROUGH THE DECOMPILER: P0 -> P1

This first attempt to translate TECO revealed some latent problems with the decompiler in areas which the ISADORA translation had not exercised. These errors were corrected in a few days, however, and the translation proceeded smoothly thereafter. Only one subroutine in TECO used an in-line calling sequence and thus needed more than the default information for its decompilation.

TECO MANUAL CHANGES: P1 -> P2

The manual changes necessary to cause decompiled TECO to compile and execute correctly can be classified into eight groups under the following headings:

1. communication with undecompiled programs,
2. skip returns,
3. self-modifying code,
4. symbol definition,
5. unimplemented instructions,
6. multiple entry points,
7. large loops or blocks, and
8. original TECO coding errors.

Table 6-D summarizes the types and number of changes needed.

Communication

TECO uses the services of a disk file system to perform file I/O (e.g., open file, read character, and close file). The disk file system was not decompiled. This led to the need for matching the contents of pseudo registers (AREG, BREG, XREG) with the machine registers for the purposes of argument passing. At the call site for the invocation of the system subroutines, code was inserted to accomplish this task. The changes were very simple.

Table 6-D. Summary of manual changes made to P1 (TECO).

Environment transitions	
subroutine calls	14
skip returns	3
Self-modifying code	
computed jump addresses	6
target addresses	2
Symbols	
wrong place	3
new symbol definitions	2
equivalent symbols	1
program block too big	3
Unimplemented code	
execute instruction	10
TECO coding errors	4
Rearrangement of code due to multiple entry points	14
Miscellaneous	3
Total number of changes	65
Number of hours	15*

* an additional 18 hours was needed to find and correct the original coding errors.

For example, the statement,

CALL CLOSE;

invoked the file close routine. This routine requires the pointer to the file control block to be in the machine A-register. The decompiler had generated the code to put the pointer in the variable AREG. The statement needed to be changed to

CALL CLOSE (AREG);.

MOL620 compiles the code to place the argument value, in this case AREG, in the machine´s A-register. Given a list of these external routines and the arguments they accept, the decompiler could perform this task automatically.

Unlike ISADORA, there were no interrupt handling routines in TECO and this reduced the environment transition modifications significantly.

Skip Returns

Some of the external routines with which TECO communicates use a special return protocol which was not present in ISADORA and was not programmed into the subroutine attribute list mechanisms of the decompiler. This protocol involves "skip returns" and is sometimes used in assembler language programming. The instruction after the call statement is itself a transfer instruction. The called routine optionally returns control to this transfer instruction or the following instruction (thus skipping over the normal return point). For example, in assembler language we might code a call to the disk READ subroutine as follows:

```
JMPM READ
JMP  EOF      ;END OF FILE RETURN POINT
  .           ;SKIP RETURN POINT
  .
  .
```

As the decompiler analyzed this code, control graph generation would proceed as if the end-of-file transfer was always taken. This would cause the EOF code to be decompiled in-line immediately after the call, as for a normal unconditional transfer. The simplest manual correction of this problem is to put two GOTO statements after the decompiled CALL. This was done for the three CALLs of this type in TECO. A change to the decompiler to include this case in its subroutine attribute list is a more general solution if this type of call occurs frequently. (See Chapter 3 for a complete discussion of subroutine attribute lists.)

Self-Modifying Code

Self modifying code appeared in TECO to compute jump addresses in the second word of a jump instruction or to set the target address in the second word of a store instruction. The decompiler discovered every case of self-modification and put warnings in the output listing which identified the location. The manual changes were easy to make.

Symbol Definitions

Three symbol definitions were moved because they appeared in the wrong place for proper MOL620 syntax. The MOL620 compiler identified these syntax errors when the program was compiled the first time. Small numeric offsets used in indexed instructions were given symbolic names.

Unimplemented Instruction Translation

Execute instructions occurred more frequently in TECO (ten times) than in ISADORA (only once). The manual changes necessary were obvious and easily made with a text editor.

Multiple Entry Points

Four subroutines in TECO had multiple entry points. Multiple entry points are not supported in MOL620 so a change was necessary. Unfortunately, the assembler language implementation of the multiple entry points involved self-modifying code and very contorted control structures. The routines were fairly short (less than thirty lines of MOL620 code) and were rewritten. Each entry point became a separate subroutine. The common code was put into a subroutine which was called by each entry procedure. This rewriting made the code much easier to understand and introduced only a small amount of overhead relative to the original.

The manual reworking of these routines was the most time-consuming of the manual changes. The automatic implementation of this translation seems to be quite difficult and probably not worth the added complexity to the decompiler.

Large Blocks of Code

The decompiler generated three program blocks inside in a huge loop (a three page WHILE statement) which caused the MOL620 compiler to exceed its compile-time stack allocations while it was translating the P2 version of TECO. The blocks were easily broken into smaller non-nested pieces. An alternative solution, increasing the working storage of the MOL620 compiler, could have been done but would have involved generating a new version of the compiler. In any case, such large nested control structures are not desirable for stylistic reasons.

Original TECO Coding Errors

I was very familiar with ISADORA and had even written some of the code in the assembler language program. There were no known bugs in ISADORA when it was decompiled. If there had been an error in ISADORA, I would have been able to trace it down quickly due to my familiarity with the system as a user and programmer. TECO was not familiar to me before I decompiled it. I had no knowledge of its internal data structures or algorithms. It had been in use for two years in our departmental computer laboratory. Because of this, I assumed that there were no known bugs and a casual survey of users confirmed my opinion.

After recompiling the P2 version of TECO, I set about to test it on the 620/i computer. Eventually I discovered three serious bugs in the editor -- two in features seldom used and one general error which could cause intermittent failure of some searching functions.

The first error found was with the EP command. This command causes the contents of a buffer to be listed on the line printer instead of a terminal. An examination of the decompiled EP subroutine immediately revealed the coding error in the assembler language version. The mnemonic code which causes the machine to output the contents of the A-register (the character to be printed) to device LPT, the line printer, is "OAR,LPT." The TECO programmer had transposed the opcode into "ORA,LPT" which means "inclusive-OR the A-register with the contents of address LPT." The decompiled code looked like

```
AREG := AREG BOR LPT;.
```

If it had been coded correctly in the original, it would have been decompiled as

```
CALL OUTPUT (AREG, @LPT);.
```

During a previous cursory inspection of the assembler version I had failed to notice the transposition of the opcode characters. This is an example of the benefit of having an alternative representation of the program. In the decompiled form the error is obvious -- the routine did not include an output statement. In the undecompiled program, the error escaped detection because the code looked right. (In early versions of the 620/i machine reference manual, this same transposition of letters occurred in a table of opcode mnemonics.)

The second TECO error encountered was much more difficult to fix. This occurred in the multiply and divide expression evaluation routine. When the test of this code failed, I examined and executed under the debugger (the new ISADORA) all of the code involved in the set up of this routine, checking for a subtle decompiler error which might have caused the problem. After a fruitless search for the problem, I happened to try this function on the production version of TECO. It did not work either and had apparently never worked. The programmer had never debugged the subroutine. It was a seldom used function, so its malfunction did not trouble the users. They just stopped trying to use it and had forgotten it didn't work. Proceeding under the knowledge that the original code was wrong, I was able to correct the problem with one new line of MOL620 code in the decompiled text.

This example suggests an obvious testing methodology. If the decompiled program does not execute correctly, check the execution of the original program to verify whether the decompiler has introduced the error or the error has simply been inherited from the original.

Finally, a third less serious TECO error was discovered while trying to discover why multiply/divide did not work. Briefly, in one routine a buffer pointer was used _before_ it was checked for validity. In the case where it was pointing to an empty buffer, it would be moved back over the contents of a contiguous data area and a searching operation using the spurious data would be done. This was not too serious since the search would "almost always" fail. Only a very low probability combination of data would cause the search on an empty buffer to succeed. This problem might not have been noticed during the two years TECO had been in service. No current user knew about it. The point to emphasize about this case is that the MOL620 version of the program made it clear that the pointer was being used before it was checked and this was not at all obvious in the assembler language version.

In this section we have indicated that the decompiler does translate logic errors into the target language along with the rest of the program. However, this experience with TECO substantiates my contention that the higher level representation does indeed aid a programmer in understanding an unfamiliar piece of software and, while the decompiler does not find the logic error in the source program, it puts the program into a form which makes the diagnosis and correction of those errors an easier task.

SIMPLE OPTIMIZING OF TECO: P2 -> P3

As with ISADORA, the majority of changes made to the P2 version of TECO involved removing redundant assignments to the pseudo accumulator, AREG. There were several cases in PO which illustrate attempts to optimize the assembler language version by the original programmer and, ironically, these attempts later caused inefficient code to be generated by the decompiler. For example, the following code sequence appeared in TECO, version PO:

(case 1)

```
MERGE 0142      ;BREG := XREG + 1
IBR             ;BREG := BREG + 1
IBR             ;BREG := BREG + 1
```

This takes three memory cycles to fetch the instructions and execute them. The decompiled code came out this way:

```
BREG := XREG + 1 + 1 + 1;
```

Recompiled without change, this would look like

(case 2)

```
LDA  XREG      ;2 cycles execution
IAR            ;1 cycle
IAR            ;1 cycle
IAR            ;1 cycle
STA  BREG      ;2 cycles
```

for a total of 7 cycles.

If the decompiler had constant expression evaluation, it might have generated:

```
BREG := XREG + 3;
```

or if MOL620 had an optimizer it would have generated the same code translation:

(case 3)

```
        LDA  XREG      ;2 cycles
        ADD  =3        ;2 cycles
        STA  BREG      ;2 cycles.
```

Here the code size is the same as case 1 (3 words), but the execution time is 6 cycles. This is because XREG, =3, and BREG are all located in memory and thus three extra memory cycles are needed as opposed to case 1 where the operands are in registers. By manually combining the constant expression 1+1+1 we arrive at the case 3 code for P3 when TECO is recompiled.

This last example illustrates the execution penalty which a decompiled program pays when the real registers of a machine are mapped into memory locations. Every reference to a real register of the source machine becomes a memory reference in the decompiled program. Coding which optimizes register usage may turn into a liability after decompilation.

A programmer writing in assembler language uses registers for four reasons:

1. economy of representation of operand address -- The memory operand often requires an extra instruction word to represent the full address, whereas the register address can be coded in a small number of bits (e.g., four bits on a sixteen register machine).

2. speed of execution -- Because a memory cycle is not needed for accessing a hard register, instruction execution speeds are greater.
3. operations available -- For example, many machines require the arithmetic accumulator to be a register. The index operation may require a register specification. On some machines, the only way to move data from one memory location to another is through a register.
4. communication protocol with other routines -- Since registers are fast global memory cells, they are often used to pass information between subroutines.

The programmer's effort which went into tuning the assembler language program register usage is completely lost in the decompilation. In fact, as the above example shows, an optimal choice of instructions in the PO program is often less than optimal in the decompiled version. The reasons for register usage are not valid in the context of a high level target language such as MOL620. In MOL620 all data items are variables. The instantiation of these variables is the concern of the compiler, not the programmer. Since the compiler cannot hope to be as clever as a human programmer, this leads to inefficiencies in the automatic translation when compared to an all out human optimization effort.

Targeting the decompilation to a more powerful machine helps alleviate this efficiency problem. For example, a sibling compiler of MOL620 is MOLSUE (Hopwood 1976). This is a compiler which accepts a language similar to MOL620 but creates code for a Lockheed Electronics SUE minicomputer. MOLSUE uses only four of the seven hardware registers of the SUE machine. The programmer can designate certain variables to be held in the remaining registers. This achieves a dramatic reduction (50 to 66%) in storage space for instructions which reference these register variables. If the target of the decompilation of 620/i programs was MOLSUE, we could naturally map the AREG, BREG, and XREG pseudo registers onto the unused registers of the SUE. The execution times of instructions involving these registers would not suffer because of the decompilation.

The simple optimizing of P2 was completed and checked out in approximately six hours. Table 6-E summarizes the number and type of changes made to the P2 version of TECO.

EXTENSIVE REWRITE OF TECO: P3 -> P4

This part of the TECO experiment was not undertaken because the benefit of having a rewritten TECO was not significant in terms of increasing program execution speed, reducing program size or providing for future evolution when compared to the time needed to complete the rewrite (estimated at fifty hours).

Table 6-E. Summary of simple manual optimizations made to TECO, P2 -> P3.

Remove redundant loads of AREG	123
Remove redundant loads of BREG	4
AREG:=AREG+1 becomes BUMP AREG	3
BREG:=BREG+1 becomes BUMP BREG	2
XREG:=XREG+1 becomes BUMP XREG	1
+1+1+1 becomes +3	3
+1+1 becomes +2	2
-1-1 becomes -2	1
Remove save of overflow status	1
Miscellaneous	5

Total number of changes	145
Number of hours	6
Number of words saved (out of 4439)	170

In the case of ISADORA, the size of the program was very important, since it occupied the same address space as the program being debugged. There were also obvious improvements which would enhance significantly the usefulness of the program. On the other hand, TECO ran alone in its memory partition. The decompiled P2 version TECO was only approximately 1600 words larger than P0. This 1600 words would come out of a TECO text buffer area of 8K words, thus reducing this area by about 20%. This reduction is not very important since TECO does not need its whole edit file in memory at one time.

Table 6-F. TECO machine code size summary.

TECO	# words(a)		expansion factor(b)		hours
Version	min	max	min	max	to create
P0 (c)	4007	4310	1.00	1.00	--
(d)	2852	3155	1.00	1.00	
P1	----- not compiled or executed -----				
P2	5642	7148	1.41	1.66	15 (e)
	4439	5945	1.56	1.88	
P3	5472	6837	1.37	1.59	6
	4269	5634	1.50	1.79	
P4	----- rewrite of P3 not attempted ----				

Notes

a. _min_ is minimum number of words in program using low core pointers and literal pool.

max is maximum number of words in program using double word instructions instead of single word instructions with low core pointers or literals.

b. min expansion factor for program Pn is calculated as

$$\frac{\text{min of version Pn}}{\text{min of version P0}}$$

max expansion factor for program Pn is calculated as

$$\frac{\text{max of version Pn}}{\text{max of version P0}}$$

c. The first line of information refers to program size including any data areas, but not the text buffer.

d. The second line refers to the program size without data area, i.e., executable code only.

e. An additional eighteen hours was needed to find the cause of the problems due to the original coding errors noted in the text of this chapter.

TECO CODE SIZE

Table 6-F summarizes the effect the decompilation had on the machine code size of the various versions of TECO.

MORE RESULTS AND COMPARATIVE DATA

This section contains more results regarding the decompilation experiments described above. When comparable data exists from the decompilation research of others, it will be discussed in relation to my own.

Manual Intervention

The manual aspects of decompiling can be classified into three stages:

1. preparation of the input data,
2. input of additional data requested by the decompiler during the translation process, and
3. modifications to the target program.

Preparation of the input data is simply assembling the source program and some minor formatting of the assembler output listing by a utility program. This text is input to the preprocessor of the decompiler. The formatted output of the preprocessor together with the source program symbol table is passed to the main routines of the decompiler. Up to this point the manual efforts consist of passing files through a set of programs. All of this preparation could be automated through the use of a command file executor.

While the analysis portion of the decompiler is operating, it may request additional information. The information supplied in these experiments was the data defining attributes of various subroutines (see Chapter 4, "Stage One Processing"). The default attributes sufficed for all but one subroutine of TECO and six of ISADORA. The information required was entered when requested by the decompiler or it was supplied beforehand as initial values of the subroutine attribute list so the decompiler could run unattended.

The manual intervention required in the first two stages is minimal. The real burden of manual intervention falls in the third stage -- modifications to the target program. The reasons for these changes were discussed earlier in this chapter. From these experiments, it appears that neither the number of changes nor the time required to make them is excessive when compared to the amount of source code involved.

One way to quantify this comparison is to determine the ratio of time needed for changes to time needed to create the original working program. Since these latter figures were not recorded by the original programmers, we estimate them for the purposes of exposition with a production rate figure of ten lines per day of debugged assembler language code. The reasonableness of this approximation is supported by the study of Boehm (1973). Assuming eight hour days and only counting lines of executable code we have the following time ratios for third stage manual changes versus initial costs

$$15/((1998/10)*8) = .0094 \text{ for TECO}$$

and

$$9/((1199/10)*8) = .0094 \text{ for ISADORA.}$$

This says that the time necessary to make manual changes to transform the output of the decompiler (P1) into an compilable and correctly executing program (P2) is really insignificant (less than 1%) relative to the time expended to create the original working program.

In his dissertation, Friedman (1974) provides a detailed report of the manual interventions necessary during his decompilation experiments. The intervention required in Friedman's experiment was done <u>before</u> decompilation -- the changes were made to the low level source program (P0). The changes made in my experiments were made <u>after</u> the decompilation to the symbolic code of the target program (P1). Certain of the changes made by Friedman were to accomodate idiosyncrasies of his decompiler or FRECL, his target language compiler. These changes are analogous to some of the changes I made to the target program (P1->P2) before compiling it with MOL620. Friedman's decompiler required knowledge of the targets of indeterminant transfers of control (e.g., indexed jumps) whereas mine did not. My decompiler treated such transfers as "out-of-window" references and assumed worst case defaults regarding variable usage and control structure information. A direct representation of all of these types of indeterminant transfers can be written in MOL620.

Table 6-G is a summary of some of the statistics we can use to compare Friedman's experiments with those reported here. The figures on manual intervention indicate that Friedman's experience was similar to mine. Relative to the number of executable source instructions in his input program, his manual intervention effort was somewhat less than mine. One would speculate that the ability to make changes after the decompilation provides a more effective method of modifying the program since the text is in a new structured form where it should be easier to understand. In addition, the decompiler and/or compiler can point out the places where the changes should be made in many cases (e.g., self-modifying code, mid-procedure entry points, and unimplemented instructions). At first glance, the statistics of these experiments do not support this contention. Friedman's time required <u>per</u> <u>change</u> is one-half of that required for a change to ISADORA and one third of that required for a change to TECO. After reflecting on this anomaly, we should remember that since my changes were made to the MOL620 version of the program, each such change would affect at least two and possibly more low level instructions. Taking this fact into account we see that the time per change per low level instruction is comparable. In any case, for the programs decompiled, the manual effort required is insignificant relative to the volume of code decompiled and so such intervention is no great barrier to transportability via decompilation.

Program Size Expansion

Of particular concern to those interested in the portability applications of decompilers is the question of how much bigger the program will get after it is decompiled and then recompiled. The code size will expand for three reasons:

1. the decompiler does not choose the best representation of the program in the target language relative to code volume (e.g., the variable usage analysis is sometimes too pessimistic),
2. the target language is not capable of representing efficiently certain operations of the low level program (e.g., nine-way parallel comparisons in one machine instruction), and
3. the compiler itself does not generate the best code that might be expected given the decompiled target program.

Table 6-G. Some comparisons of the TECO and ISADORA experiments with Friedman's data.

	TECO	ISADORA	Friedman
No. of executable source instructions(PO)	1998	1199	4863(a)
No. of manual changes	65	55	288(b)
Time required for changes (hours)	15	9	24(c)
Time required per change (hours)	.23	.16	.08
No. of changes/instruction	.033	.046	.059
No. of executable instructions after decompiling/recompiling	3369	2231	14297(a)
Expansion factor due to whole system	1.67	1.86	2.94(a)
Expansion factor due to target language and compiler	1.25	1.25	2.12(d)
Expansion factor due to decompiler alone	1.34	1.49	1.39(a)

Notes:

a. from Friedman (1974), page 143, Table 5D.

b. from Friedman (1974), page 136, Table 5B and personal correspondence (Hopwood and Friedman 1976).

c. from Friedman (1974), page 141.

d. the expansion factors are related as follows: The total expansion factor of the transport system is the product of the expansion factors for the decompiler and the target language compiler (e.g., 2.94 = 1.39 * 2.12).

The last two points are very much related. Where to put the blame for inefficiencies in object code created by a compiler is often unclear. Is the inefficiency the fault of the language design or of a lack of optimization capability in the compiler?

The number of executable statements in the machine language program resulting from compiling the target program output by the decompiler is recorded in Table 6-G. In the case of TECO and ISADORA, the P2 MOL620 program was compiled into 620/i machine code. In the case of Friedman's program the, the FRECL compiler produced Microdata 1621 machine code. Since Friedman's statistics are presented in terms of the number of executable instructions rather than the size of the executable code in storage units (program minus data), I have done the same for comparison purposes. The expansion factor, the ratio of the number of executable statements after decompilation/recompilation compared with the number before decompilation, shows rather marked differences between the two systems (1.67 for TECO and 1.86 for ISADORA versus 2.94 for Friedman's program).

These figures may be slightly misleading and should be interpreted with care. Table 5-D of Friedman's dissertation (p. 143) contains information regarding program expansion. The corrected version of that table was compiled (Hopwood and Friedman 1976) and is reproduced here in Table 6-H for the interest of any reader who may wish to read Friedman's work. Friedman attempts to quantify the reasons for the 194% increase in program size (executable statements). The table is supported by several pages of explanation which can be summarized as follows. The expansion factor due to the decompiler itself is about 1.39. The expansion factor due to the FRECL language and compiler relative to the Microdata instruction set is about 2.12. The product of these two factors is the final expansion factor 2.94.

Table 6-H. Friedman's corrected Table 5D.

Program Size Increase Data
for the OS/1621 Experiment

(1)	Total number of executable statements (ES) in the original 32 OS/1621 programs	4863
(2)	Total number of ES in the transported OS/1621 programs	14297
(3)	Total ES increase: (2)-(1)	9434
(4)	Percent increase in ES: (3)/(1)	194
(5)	Percent of the total ES increase (3) caused by Microdata instruction set asymmetries	53
(6)	Percent of the total ES increase (3) caused by inefficient code generation for computations	27
(7)	Percent of the total ES increase (3) from causes other than (5) and (6)	20
(8)	Total ES increase from causes other than (5) and (6): 20%*(3)	1887
(9)	Percent increase in ES from sources other than (5) and (6): (8)/(1)	39

Note--the 194% figure (line 4) corresponds to a 2.94 expansion factor.

From my experience with MOL620 over a six year period, a typical expansion factor due to the language and compiler is about 1.25. (This is the same figure for the expansion factor of the P4 version of ISADORA , the complete rewrite. An expansion factor of 1.25 says that one can expect a typical MOL620 program to produce code about 25% larger than a program written to do the same thing in assembler language.) Using this language/compiler expansion factor of 1.25 we have a decompiler expansion factor of 1.34 for TECO and 1.49 for ISADORA. Of course, these estimates of the expansion factors due to the target languages and compilers are just that -- estimates. However, they do give us a chance to isolate the decompiler inefficiencies and show the sensitivity of program size expansion due to each part of a decompiler based transporting system.

The differences in expansion factors of the language and compiler for my system (1.25) versus Friedman's (2.12) can be explained in the following way. MOL620 was a very mature system with a significant portion of its code generator devoted to optimization of instruction selection. FRECL was a recently built system which had not evolved to a point where it could cope successfully with the asymmetries in the Microdata instruction set or perform much optimization. Given several more years of evolution one could reasonably expect the efficiency of the FRECL compiler to improve.

It is interesting to see that the expansion factor estimates for the two decompilers are very similar. My experiment optimizing P2 versions of TECO and ISADORA suggests that the expansion factor of my decompiler cannot be improved much more without a very substantial and sophisticated redesign of the system. As indicated by the simple optimization experiment, a 10% reduction of the decompiler expansion factor to 1.20 seems the limit point for this decompiler technology. Although Friedman did not predict this figure I would expect the same to be true of his decompiler as well.

Decompilation Speed

During the course of these experiments I have recorded the time necessary to perform the decompilation of TECO and ISADORA. Table 6-I summarizes the execution time statistics. As the note on the figure suggests, these times could be reduced by compiling the LISP code of the decompiler rather than interpreting it. Friedman does not comment on the speed of his decompiler but Housel (1973, p. 192) reports a decompilation speed of 1080 source instructions per minute on a CDC 6500 using about 23K words of memory. These figures were gathered from experiments decompiling very small programs (less than 100 statements). Housel´s decompiler was written in FORTRAN which compiled into machine code which was then executed directly by the

CDC 6500. Friedman used the Housel decompiler as a base to build his own system so we should expect similar speeds. (It should be noted here that the Housel/Friedman decompiler does not preserve source program comments in the target program. The processing of comments consumes some cpu time and a portion of the dynamic data area.)

Table 6-I. Execution speed of the decompiler.

	TECO	ISADORA
No. of source lines	3211	2041
No. of executable instructions	1998	1199
Main memory utilization (1024 36-bit words)	68	55
Total cpu time (hours)	1.32	.56
Source lines/min	40.5	60.7
Executable instructions/min	25.2	35.7

The cpu time was for a DECsystem-10 KI processor with two microsecond memory executing the LISP interpretive system. Compiling the decompiler should yield execution speed improvements of a least 10:1, and possibly 20:1 in certain cases. In addition, compiled code is smaller.

There are other reasons why the decompiler presented here is slow. In addition to being interpreted, it ran in a virtual memory system where the execution time of the page fault handler is charged to the user who faults. In a heavily loaded system the page fault rate dramatically increases and the observed speed of the decompiler is reduced. This fact explains the difference in the decompilation rates of TECO and ISADORA shown in Table 6-I. TECO was decompiled during a time of very heavy load on the PDP-10 system while ISADORA was decompiled at a time when the system was lightly loaded. TECO required more pages of memory than ISADORA and this increased the paging rate of the decompiler.

Table 6-J lists the proportions of execution time spent in each of the decompiler activities (except pre- and post-processing time, which was minimal). Of the total time recorded in Table 6-I, 44% was overhead time -- I/O time, garbage collection time, page faulting, or time spent in common general purpose shared routines. The remaining 56% of the cpu time was divided into five activities -- stage one control graph generation, stage two modifications to the control graph, IMTEXT generation, forward substitution, and target program code generation. (See Chapter 4 for details about each of these activities.) Table 5-A shows the breakdown of sizes of the decompiler components. If we spread the overhead routines proportionally over the five activities, the relative cpu times used by each activity are roughly the same as their relative code volumes.

Table 6-J. Proportion of non-overhead cpu execution time and code size for each decompiler activity.

	Execution Time	Code Size
Stage ~~One~~ Processing	7.8%	9.1%
Stage Two Processing	4.4%	8.4%
IMTEXT Generation	23.3%	28.7%
Forward Substitution	20.1%	14.9%
Target Code Generation	44.8%	38.9%

Overhead time (I/O, garbage collection, etc.) accounted for 44% of the total cpu time. The activities above accounted for the rest.

Source vs. Target Program Text

It is of some interest to compare the language constructs appearing in the source and target program texts. In particular, what are the target statements created by the decompiler and how do they relate to the source instructions? From the cross reference listing output by the 620/i assembler I was able to derive the frequency of use of instructions in the source assembler language programs. This information is listed in Table 6-K. I instrumented the MOL620 compiler so that it would count the number of each kind of high level statement it translated. This information is listed in Table 6-L.

Two important questions can be answered with this data. How well does the control structure recognition and generation work? How well does the expression condensation algorithm work?

Control Structure. The occurrence of jump instructions and jump targets (indicated by labelled instructions) are the significant symbolic keys to construction of the control graph of the program. An unconditional jump corresponds to a break in the normal sequence of processing. A conditional jump corresponds to a fork in the instruction stream. A labelled instruction corresponds to a join of two or more control paths.

There are several different target language constructs that might be generated due to a jump instruction appearing in the instruction stream analyzed by the decompiler. These are listed below:

unconditional direct	GOTO direct WHILE stmt

conditional direct	IF exp THEN GOTO direct
	IF exp THEN stmt
	WHILE stmt

unconditional indirect	GOTO indirect RETURN

conditional indirect	IF exp THEN GOTO indirect
	IF exp THEN RETURN

Table 6-L lists the symbolic language constructs actually generated from the source program. The number of GOTO statements appearing in the target program represent a 62% ((714-269)/714) reduction in the number of the assembly language analogs in the source program. Statement labels were reduced by 68% ((370-118)/370). If we discount the forty-six (46) labels which appear in the error routine of

TECO which cannot be removed under any reasonable rewriting scheme, we have a reduction of 78% instead.

Where did the GOTOs go? They were replaced by the control statements -- IF...THEN, IF...THEN...ELSE, RETURN, and WHILE. The most difficult structures to recognize were the IF statements with a block body and the WHILE statement. The IF...THEN...ELSE construct accounted for only 17% (34/(34+171)) of the non-simple IF statements generated. Unlike the Housel/Friedman decompiler, my decompiler attempts to avoid unnecessary levels of block indentation by decomposing an IF...THEN...ELSE into two simpler statements by using the following transformations:

```
   IF E THEN...GOTO L ELSE S;  => IF E THEN...GOTO L; S;
 IF E THEN S ELSE...GOTO L;  => IF NOT E THEN...GOTO L; S;
```

The WHILE TRUE form of the while loop corresponds to a loop with a test which does not occur at the beginning of the loop. Mid- or post-test loops were not supported in MOL620 so the WHILE TRUE form was generated with an explicit break from the interior of the loop by a GOTO or RETURN. Only 24% (12/51) of the while loops generated fit the pre-test pattern. This seems to indicate that an attempt to generate other types of control structures such as REPEAT statements would lead to a greater fraction of the loops fitting a standard pattern. From a brief analysis of the situations where the mid- and post-test loops are used, a possible reorganization of the ordering of conditional tests, generation of synthetic control variables or node splitting (see Chapter 3) could be used to force the loop into a pre-test or post-test form. In some cases, this transformation would improve the clarity of the code, in other cases the program would be more difficult to understand.

Expression Condensation. Table 6-K records the number of source program instructions which cause movement or operations upon data in registers or memory. In a simple-minded decompiler with no ability to combine components of expressions, we would expect the number of assignment statements in the target program to equal the number of data movement/operation instructions in the source program. For example, we might expect the assembler language sequence

```
LDA  M
SUB  N
STA  P
```

to generate the target instructions

```
AREG := M;
AREG := AREG - N;
P := AREG;
```

instead of the more appropriate statement

```
P := M - N;
```

Table 6-L indicates that the decompiler developed in this research has achieved a 52% ((1959-945)/1959) reduction in the number of assignment statements that might have been expected if the expression condensation mechanisms were not active. (This figure is 49% if we choose to remove the INR opcode from the figures, since this is translated directly to the MOL620 BUMP statement and no forward substitution is done on it.) This kind of improvement in the generation of target expressions is well worth the effort invested in the construction of the expression condensation forward substitution algorithms (about 11% of the total decompiler code). Expression condensation makes explicit the interdependencies of subexpression elements. Absence of this capability (as in the Ultrasystems decompiler discussed in Chapter 2) severely degrades the quality of a target program when measured in terms of readability.

Comments and Formatting

As described in Chapter 3, the symbolic source code of the program to be decompiled provides much information not present in the assembled binary object file. The comments written with a program are valuable when it comes to understanding what the program is doing. The decompiler built in this research moves the comments to the target program. If the comment was near a source instruction in the original program, it will appear near the target statement generated for that source instruction.

Table 6-K. Frequency of symbolic language constructs in the TECO and ISADORA source programs (PO).

	TECO	ISADORA	Total
Source Lines	3211	2041	5252
comment lines	629	523	1152
listing control	64	49	113
non-code defns	142	70	212
executable insts	1998	1199	3197
data declarations	378	200	578
Jump instructions	447	267	714
uncondtl direct	108	64	172
uncondtl indirect	101	60	161
condtl direct	217	132	349
condtl indirect	21	11	32
Subroutine call insts	278	207	485
uncondtl direct	255	196	451
uncondtl indirect	20	5	25
condtl direct	3	6	9
condtl indirect	0	0	0
Data Move/operate insts	1252	707	1959
load register	458	250	708
store register	270	147	417
others(add,sub,...)	449	283	732
incr/replace(INR)	75	27	102
I/O instructions	1	12	13
Subroutines	99	73	172
Instruction labels	273	97	370

Table 6-L. Frequency of symbolic constructs in the TECO and ISADORA target programs (P2).

	TECO	ISADORA	Total
Source lines (a)	3863	2581	6444
Executable statements	1375	940	2315
Non-executable stmts	409	148	557
definitions	99	22	121
declarations	310	136	446
IF statements	252	146	398
IF...THEN...ELSE	19	15	34
IF...THEN GOTO	106	40	146
IF...THEN CALL	7	8	15
IF...THEN RETURN	20	12	32
IF...THEN (others)	100	71	171
WHILE statements	26	25	51
WHILE TRUE...	20	19	39
WHILE exp...	6	6	12
BEGIN...END blocks	123	90	213
Assignment statements	563	382	945
Assignment expressions	373	164	537
GOTO statements (b)	185	84	269
direct	172	73	245
indirect	13	11	24
CALL statements (b)	306	248	554
direct	288	240	528
indirect	18	8	26
RETURN statements (b)	107	64	171
BUMP statements	82	27	109
Procedures	145	100	245
Referenced stmt labels	97 (c)	21	118

Notes:

a. MOL620 statements are free form and may occupy several lines of text.
b. includes counts for IF...THEN GOTO, CALL, RETURN, resp.
c. includes 46 counts for labels in TECO error routine.

This feature is lacking in all of the true decompilers surveyed in this research. Since I estimate that it only took about 5% of the total implementation effort to provide this facility, it should be included in every such system which reads symbolic code.

The formatting of the target program by the post-processor is important to the readability of the program, e.g., indentation of blocks, alignment of comments, blank lines, labels placed at the left margin. These formatting conventions are stylistic rearrangements of a free format program. The rearrangements provide a visual exposition of the logical structure of the program when it is presented to the human eye.

SUMMARY

In this chapter, we have presented the results of decompiling two production programs, TECO and ISADORA. These programs were decompiled and then recompiled and successfully executed. Necessary manual interventions, code volume expansions, decompilation speeds, and language constructs were analyzed. Where comparative figures existed, they have been presented. The reader is referred to Appendices IV and V to examine excerpts from the P0 and P2 versions of ISADORA.

Chapter 7

SUMMARY AND CONCLUSIONS

INTRODUCTION

This work is an in-depth analysis of a translation process called decompilation. Previous studies of this process are summarized in Chapter 2. Chapter 3 is a guide for others interested in building their own decompilation systems. The suggestions and observations contained in Chapter 3 were derived from the actual design and implementation of the decompilation system described in Chapters 4 and 5. The results of the decompilation experiments reported in Chapter 6 provide the most complete set of measures available on the performance of a decompiler and the first empirical demonstration that large production programs for minicomputers can be decompiled then recompiled so that they execute properly and with reasonable efficiency. The subjective notions of readability and understandability have been addressed in the description of the experiments in Chapter 6. The decompiled program did prove valuable in debugging the target programs and was the preferred reference text after the decompilation.

Together with the earlier studies of decompilation (especially, Barbe, Housel, Friedman, and Ikezawa), this work should prove to be a valuable reference volume to others interested in decompilation. It represents a documented successful attempt to solve the decompilation problem in an important subset of the problem domain. It shows that much can be accomplished despite the very difficult problems of transferability (portability) discussed in Chapter 3. Others should now have a better idea of what to expect when they embark on the task of building a decompiler.

In the rest of this chapter some important pragmatic considerations which have been discussed in earlier chapters (especially Chapter 6) will be discussed. We will also discuss the expected uses of decompilers and their viability as tools in the computing community, future decompiler research directions, and the immediate implications of this research.

PRAGMATICS

The first questions about decompilers from people interested in its pragmatic applications usually focus upon the following:

- o What languages can be decompiled? To what targets?
- o How much manual intervention is necessary?
- o How efficient is the decompiler? What is the size and speed of the decompiled program relative to the

original? How fast does the decompiler execute?

o What does a decompiler system look like? How much does it cost to build?

These questions are answered in detail in the body of this work and the conclusions are summarized below.

<u>Source-Target Pairs</u>

Those who work on decompilation systems believe that the best results can be achieved by restricting the class of source-target language pairs to those languages which are compatible in most respects, i.e., the language constructs appearing in the source program must have a rather simple analog in the target language. If the source language is assembler language, the class of languages most suitable as targets are machine oriented languages, such as the MOL620 language used in this research. As the name implies, these languages are designed as efficient vehicles to represent algorithms on a particular machine or machine family (see van der Poel and Maarsen 1974).

Our experience with machine oriented languages for minicomputers (MOL620, designed in 1969, was one of the first for a minicomputer) indicates that programs written in machine oriented languages are often quite portable and amenable to automatic translation to other minicomputers. The de facto sixteen bit word length standard and the rather limited instruction sets of minicomputers are the two most important reasons for this compatibility. One might argue that a more desirable target language for decompilation would be a standard machine independent language such as FORTRAN or COBOL. Some commercial companies offer such services. The customer can be expected to pay for this standardization with increased manual intervention costs, unstructured code, larger target programs and run-time inefficiencies. The costs for commercial decompilation are often priced at several levels. The lowest cost service is automatic translation only. The highest cost service is complete translation and modification to make the target program work according to a set of specifications. As we have seen, the later service often requires manual intervention and testing and may involve rewriting certain portions of the program.

Manual Intervention

When the target language adequately covers the source language, as is the case in the experiments presented here, the manual intervention necessary during the decompilation process is minimal. This work and that of Friedman have quantified the manual effort involved. I would predict that the effort required for manual intervention is proportional to the difference between the source and target languages and the source and target machines. For example, I would expect that it would have taken significantly more manual effort to rework the decompiled output into a legal FORTRAN program rather that the machine oriented language which was used.

Friedman (1974) reports on an effort to translate an IBM 1130 Disk Monitor System (DMS) program into FRECL, his machine oriented language for the Microdata 1621. On page 134 of his dissertation he states:

> "...while it should be possible to transport the IBM 1130 DMS to ... the Microdata, the overall effort that would be required to prepare the DMS system code for input to the Transport System was for too great to be completed in a reasonable amount of time. It seemed more could be learned by using the Transport System to obtain the FRECL representations of operating systems code for one computer, and recompiling this code back to the assembly language of the same computer."

He abandoned the 1130 effort and concentrated on the decompilation of the 1621 code. The match between the 1130 virtual machine and the FRECL virtual machine was not close. For similar reasons, I rejected the notion of translating 620/i programs to FORTRAN or PL/I -- the projected manual effort involved was too great. The machine oriented language approach seemed to the the best compromise. Having already written three machine oriented language compilers, I knew that in the class of sixteen bit minicomputers, these languages are often more portable than their names suggest. Experience with the Distributed Computing System (Farber et.al. 1973), a network utilizing three different minicomputers demonstrated this fact. Software written for one machine could be rewritten in a short period of time for another machine using a similar language dialect. The different dialects emphasized the architectural strengths of the machines for which they were designed, but they maintained a uniformity of syntax and semantics with respect to machine independent concepts such as expression denotation and control structure syntax.

Code Volume

For the purposes of transferability, the expansion of code volume caused by the decompilation process is an important consideration. The match between the source and target machine (language and interpreter) must be close to prevent excessive inflation of program size as well as other inefficiencies.

In the ISADORA experiment, an expansion ratio of 10:1 for overflow testing instruction translation was seen. The match between the 620/i machine language and MOL620 was not close in that case. The expansion factor for the total ISADORA experiment ranged from 1.51 to 2.02 (see Table 6-C), and from 1.37 to 1.88 (see Table 6-F) for TECO. Part of this expansion was due to the decompiler and part due to the MOL620 language and compiler. The inefficiencies present in the target program are a result of the product of the inefficiencies of each translation step involved. This observation suggests that efforts to reduce the size of decompiled code should be directed at every translator in the chain, not only at the decompiler itself.

The importance of the code size expansion problem relative to other aspects of the decompilation process such as portability, understandability, or language level can only be decided in the context of a particular application. For example, if transferring a program via decompilation results in a program which cannot execute on the target machine because it is too big, then the use of the decompiler to achieve the transfer is not the proper approach. On the other hand, if the target program's higher level text is to serve as documentation for a redesign of the program, then the fact that the decompiled output is too large to execute is irrelevant.

Our experiments show that the observed expansion factor is sufficiently small that it will be acceptable in a large number of contexts. Moreover, the space a program occupies is rapidly becoming a secondary consideration in the light of decreasing main memory costs and increasing software production costs. The importance of code size expansion should tend to diminish for many applications in the coming years.

Speed of Decompiler Execution

The speed of a decompiler is not nearly as important as the speed of a compiler, since a given program will not be decompiled more than once or twice for transfer to another language. In its eventual evolved production form, a decompiler similar to the one implemented in this research should begin to approach the speed of an optimizing compiler since many of the control structures and variable usage analysis algorithms are similar (see Chapter 6 for comparative results).

Speed of the Target Program

The largest execution penalty in target program execution is due to the mapping of variables from fast access memory (e.g., registers) to slow main memory. Control transfer instructions should normally incur no degradation in execution time. Certain machine specific operations (such as overflow testing in the 620/i) which are not easily represented in the target language, will incur a higher penalty in execution speed since many more instructions will be needed in the target language to simulate the effect. As with program expansion, execution speed degradation is partly effected by the target language and target compiler as well as the decompiler. In this research, execution speeds of the decompiled programs were not noticeably different to a user of the target program even though the target program was averaging about three memory cycles in the target program for every one in the source. These programs were interactive and I/O bound in execution.

Cost of Building a Decompiler

The design of the decompiler built in this research is outlined in detail in Chapter 4. The designs of other decompilers have been discussed in Chapter 2. Discounting the research and design time, the cost of building a decompiler can be compared to that of building a good optimizing compiler. Chapter 5 summarizes the amount of code in each part of the decompiler in Table 5-A. Of course, the effort per program unit differed depending upon the task performed. For example, the effort expended per line of code in designing and testing the forward substitution algorithms was much greater than that expended per line of service routine code. This difference is due to the relative complexity of the algorithms. The amount of time spent designing and implementing the decompiler was not recorded, but an estimate of one man year to build such a translator should be approximately correct. Given that decompiler designs are now available in the literature, effort can be concentrated on refinement and implementation enhancements rather than research into new methods.

Choice of language and computing system environments in which to write the decompiler itself is important and can significantly reduce or increase the amount of time needed to produce a working system. LISP and the DECsystem-10 have proven to be a good choice in this regard. Friedman (1974b) has indicated FORTRAN on the CDC 6600 is not a good choice.

EXPECTED USES OF DECOMPILERS

The expected uses of decompilers can be divided into two main categories (Hopwood 1976):

1. transfer of software to other languages and machines, and
2. aids to understanding software -- as documentation, for validation and verification, and for static analysis of program features.

Transferability

The problem of manual intervention is obviously closely related to that of transferablity. However, the question of transferability can be separated from that of machine independence. Transferring software from a particular machine, M1, to another machine, M2, is a much easier problem that the translation of M1 code into a universal machine independent language. The universal language does not exist and is not likely to ever exist. (See Steel 1966 for more on UNCOL). With a particular target machine in mind, the difficulties encountered arise because of incompatibility between the source and target language interpreters. Purely syntactic problems in the languages are easily handled, the semantics are the problem. Chapter 3 addresses many of these difficulties in detail.

The most important task to be performed by an analyst considering the use of a decompiler for transporting software is to study the costs imposed on the translation by the source/target differences. Assuming the source language is given, it appears that a judicious choice of a target language can reduce the cost of the translation and improve the chances for satisfactory performance of the target software along all dimensions. The widely recognized and implemented languages such as FORTRAN, COBOL, ALGOL 60, and PL/I are probably the initial candidates one would consider for the target language. Housel (1973) translated to PL/I and Hollander (1973) to a dialect of ALGOL. Sassaman (1966) targeted to FORTRAN, Halstead (1962) to NELIAC, and several commercial vendors advertise an IBM 1400 to COBOL conversion. For the decompilation effort which does not require machine independence, the many machine oriented languages now in existence provide a suitable target for moving the language level of a program up within a single machine architecture family. The greater efficiency of these languages relative to the machine independent ones is a powerful incentive to choose this alternative (see van der Poel and Maarsen 1974).

Documentation Aid

The use of decompiled output to replace or supplement traditional forms of documentation has been suggested (Hopwood 1974). The Ultrasystems decompiler mentioned in Chapter 2 was built for this purpose. In my opinion, the use of decompilers for the primary purpose of documentation will be minimal for several reasons. In the future, programming in assembler language will become the exception rather than the rule. Higher level languages will be used for almost all of the purposes for which low level languages are now used. Old programs will be discarded in favor of more modern replacements or will be translated into higher level languages by hand or by automatic aids such as decompilers. Only in rare cases will the assembler language program be retained with the decompiled version used only for documentation.

Evolutionary Path

The most promising use of a decompiler is for the purpose of modernizing a body of assembler language software where the original code is used as a reference model for implementation of algorithms which should be retained in the new version. In cases where the code must be very efficient, the decompiled code can be used as a guide to a complete manual rewrite.

Validation and Verification

As the work of Ikezawa (see Chapter 2) suggests, another very promising area of decompiler application will be in the field of software validation and verification. The analysis which a decompiler performs, combined with modern verification techniques, provides a valuable tool for program inspection.

Static Analysis

The analysis algorithms of a decompiler can be used to gather statistics about interesting features of programs. Examples of such information include the location of dead code areas, number of different looping conventions used, frequency and type of instruction modification, busy status of variables, scope of conditional control and exit conditions from subroutines. This information is useful in debugging, validation and verification, and program transformations of low level source code.

FUTURE RESEARCH DIRECTIONS

Data Structure Abstraction

An open area of research is the abstraction of data structure from low level program descriptions. The use of target languages such as Pascal (Jensen and Wirth 1975) or PL/I demands more sophistication from the decompiler for the description of data structures than the simple transliteration of assembler language definitions used in this research. For example, assume an assembler language program uses a data structure defined implicitly by symbolic equates and instruction usage to correspond to the Pascal record definition:

```
RECORD
     NAME : ARRAY[1..15] OF CHAR;
     ID : 0..9999;       (*FOUR DIGITS*)
     HEIGHT : 36..96;          (*INCHES*)
     WEIGHT : 50..400;         (*POUNDS*)
     SEX    : (FEMALE, MALE);
     CITIZEN : BOOLEAN
END;.
```

A programmer might deduce such a definition by examining how the program accesses data, but the methods of automating such analysis are far from understood. Housel (1973) describes an attempt to determine the extent of arrays based upon their indexed references. This is about as far as any decompiler has gone and the results are still quite primitive. It seems much more effective to simply look at the symbolic storage reservation pseudo operations of the source language program. This was the approach taken in this research. This works well because the target machine is very similar to the source machine.

More research into the methods of abstraction of data structures from program referencing is needed. Some of the work of those interested in using data abstraction to produce high level program descriptions may prove helpful in this area (see Rowe 1976).

Control Structure Recognition and Transformation

The main emphasis of this and previous decompilation research is on the analysis of program control structure. This research, and that of others reported here (except Hollander) use a procedural oriented method of discovering patterns in the control structure of the source program being decompiled. For example, a procedure in an algorithm~~ic~~ language (e.g., LISP) is written to find a WHILE loop pattern, or an IF...THEN...ELSE pattern.

Hollander (1973) used the pattern matching facilities of a syntax directed parser. This method is weak because it cannot recognize semantically equivalent but syntactically different patterns. The work of M.D. Hopwood (1974) suggests an alternative method of pattern matching based upon a set of tree transformations applied to a semantic tree representation of the source program. A set of patterns and transformations would replace the combinatorial logic of special purpose procedures and would serve to standardize this section of the decompiler.

High Level Source-to-Source Transformations

The work of Standish (1976) and Loveman (1977) suggests that programs expressed in higher level languages may be improved by application of rewriting rules possibly guided by humans. The use of a decompiler to translate valuable assembler language programs into the higher level working language for further automated refinement would be an interesting marriage of the two systems.

Software Physics

The work of Halstead and Bayer (1972, 1973) and Bulut et.al. (1974) suggest a theory of the thermodynamics of algorithms, an information theoretic approach to the quantification of such programming terms as "program volume," "language level," and "quality."

The source and target versions of a decompiled program are different representations of the same algorithm. What can be said about the effect of decompilation upon the Halstead measures? Does program volume go up? Does language level and internal quality increase? A set of experiments dealing with these questions might lead to a standard method of measuring the performance of a decompiler. In addition, since the source and target algorithms remain the same, and only the language representation is different, the elimination of this variable might lead to interesting validation of the software physics theory itself. Do the measures agree with what we believe is happening to the decompiled program?

The work (in thermodynamic terms) a decompiler must do in order to discover certain relationships between variables or control paths should be quantifiable in terms of software physics. Research in this area might lead to a better understanding of why certain programs are easier to understand than others, i.e., a quantification of the meaning of good programming style. If style can be measured, then much greater automatic quality control can be exercised over the style of programs produced in the software factories of the future. The benefits in the reduction of maintenance costs are obvious.

IMMEDIATE IMPLICATIONS

Structured Coding Techniques Using Low Level Languages

Decompilation research has shown that low level programs can be translated into higher level languages. The techniques used for such translations suggest an assembler language coding methodology to that proposed for the higher level languages, e.g., standardized control mechanisms and restricted variable scope. Those software organizations which persist in using low level languages would do well to apply such standards to their programming efforts to assure consistent, easy to understand, and decompilable programs.

Software Evolution

The decline of assembler language as a primary programming language has begun. The declining cost of hardware and the increasing costs of labor intensive products assure that only the most primitive machines will be programmed in low level languages. The trend is clearly seen in the medium and large scale machine environments. In the minicomputer environment, the typical machine configuration of today is more powerful by almost any measure than the largest machines of the late 1950's. The minicomputer manufacturers are faced with similar software development and maintenance problems similar to those of mainframers and are gradually moving away from the use of assembler languages for system development. Microcomputers are now being almost exclusively programmed in assembler languages. Higher level languages, however, are being designed for use on such small systems.

As manufacturers of all size machines begin to make use of higher level languages in the next few years, we can predict a resurgence of interest in decompilers similar to that which accompanied the transition to the third generation of computers in the mid 1960's. Companies will wish to save some of the software investment represented by their assembler language libraries. Many of the older programs will be obsolete and not worth carrying forward into a new language, but some, particularly application programs such as editors, compilers, and file systems might be decompiled. Modernization of a software product might very well call for the use of a decompiler as part of the evolutionary process.

The manufacturers of small machines can afford to produce hardware closer to the current limits of the technology than the larger mainframe manufacturers who only introduce new central processor products every few years. This fact will lead to the introduction of new machine architectures which must be programmed either in a new assembler language or a higher level language. The obvious choice is to begin use of a higher level language at the introduction of a new machine line. In preparation for that transition, the good programs from an older line can be converted to the new system implementation language with the help of decompilers.

CONCLUSION

The discussion of the benefits of decompilation must necessarily be based upon subjective notions of what comprises good program presentation (style) and how that presentation is achieved, as well as the more pragmatic considerations of the importance of code size expansion, costs of decompilations, and uses of the final product.

In the earlier discussions of the two experiments conducted in this research, examples were given of why and how the decompiled target program text was a more useful representation of the program than the original assembler language text. The decompiled program is more useful because its control structures and expression evaluation structures are more visible when expressed in the higher level language. Although the decompiler does not attempt to abstract data structures, the data structure is more easily comprehended when viewed through the abstracted control and evaluation structures. This last point was made clear while attempting to debug the P2 version of the TECO program.

Although the decompiled programs are easier to understand, it is clear that they would be considered to be of low quality if produced by a competent professional programmer. This simply reflects the fact that the original programs were not very well structured or implemented. A decompiler reveals the structure of the source program. It cannot rewrite it, except in trivial automatic ways, e.g., node splitting. It cannot turn a sow's ear into a silk purse. If we put low level garbage into the decompiler, we are very likely to produce high level garbage. To be sure, the latter is better than the former, but is far from our idea of what the ultimate decompiler should do.

In pragmatic terms, the experiments of this research have shown that decompilation can be a useful aid to evolutionary program improvement. This was illustrated in the case of ISADORA when the target program was substantially enhanced after decompilation produced a higher level language version. This means that a valuable production program can be transformed over a period of time from a poorly structured low level language implementation to a well structured high level language program. The decompiler is a tool to break through the initial barrier of language level. A human continues the process.

References

Aho, A.V. and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

Allen, F.E. and J. Cocke. "A Program Data Flow Analysis Procedure." Communications of the Association for Computing Machinery, Vol. 19, No. 3, March 1976, pp. 137-147.

Barbe, P. "Techniques for Automatic Program Translation." Software Engineering, Vol. 1. Ed. J.T. Tou. New york: Academic Press, 1969.

Barbe, P. "The PILER System of Computer Program Translation." NTIS Report No. AD/A-000294/9. Probe Consultants Inc., Phoenix, Arizona, September 1974.

Bobrow, R.J., R.R. Burton, J.M. Jacobs, and D. Lewis. "UCI LISP Manual." Department of Information and Computer Science, University of California, Irvine, Technical Report #21, October 1973.

Boehm, B.W. "Software and Its Impact: A Quantitative Assessment." Datamation, May 1973, pp. 48-59

Breuer, M.A., ed. Design Automation of Digital Systems, Vol. 1, Theory and Techniques. Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

Bulut, N., M.H. Halstead and R. Bayer. "Experimental Validation of a Structural Property of FORTRAN Programs." Proceedings of the Annual Conference of the ACM, November 1974, pp. 207-211.

Climenson, W.D. "Irreducible Flowcharts." Department of Information and Computer Science, University of California, Irvine, Technical Report #35, April 1973.

de Balbine, G. "Better Manpower Utilization Using Automatic Restructuring." Proceedings of the National Computer Conference, Vol. 44, 1975, pp. 319-327.

Farber, D.J., J. Feldman, F.R. Heinrich, M.D. Hopwood, K.C. Larson, and L.A.Rowe. "The Distributed Computing System." Proceedings of the Seventh Annual IEEE Computer Society International Conference, February 1973, pp. 31-34.

Feldman, J.A. and D. Gries. "Translator Writing Systems." Communications of the Association for Computing Machinery, Vol. 11, No. 2, February 1968, pp. 77-113.

Friedman, F.L. Decompilation and the Transfer of Mini-Computer Operating Systems. Ph.D. Dissertation. Department of Computer Science, Purdue University, August 1974.

Friedman, F.L. Personal communications. May 1974b.

Gaines, R.S. "On the Translation of Machine Language Programs." Communications of the Association for Computing Machinery, Vol. 8, No. 12, December 1965, pp. 736-741.

Griswold, R.E., J.F. Poage, and I.P. Polonsky. The SNOBOL4 Programming Language, second edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1971.

Halstead, M.H. and R. Bayer. "Algorithm Dynamics." Proceedings of the Annual Conference of the ACM, Atlanta, Georgia, 1973, pp. 125-135.

Halstead, M.H. Machine Independent Computer Programming. Washington, D.C.: Spartan Books, 1962.

Halstead, M.H. "Machine Independence and Third Generation Computers." Proceedings of the Spring Joint Computer Conference, 1967, pp. 587-592.

Halstead, M.H. "Using the Computer for Program Conversion." Datamation, May 1970, pp. 125-129.

Hollander, C.R. Decompilation of Object Programs Ph.D. Dissertation. Digital Systems Laboratory, Stanford Electronics Laboratories, Stanford University, SEL-73-029, Technical Report No. 54, January 1973.

Hopwood, G.L. "Inverse Compiling for Program Documentation." Oral presentation, Abstract in Proceedings of the Annual Conference of the ACM, San Diego, California, November 1974, p. 751.

Hopwood, G.L. "Decompilers -- Tools or Toys?" Oral presentation, ACM Computer Science Conference, Anaheim, California, February 10-12, 1976.

Hopwood, G.L. "MOLSUE: A Machine-Oriented System Implementation Language for the Lockheed Electronics SUE Computer. Language Reference Manual." Department of Information and Computer Science, University of California, Irvine, Technical Report #86, March 1976 (revised February 1977).

Hopwood, G.L. and F. Friedman. Personal correspondence. March, April 1976.

Hopwood, M.D. A Semantic Formalism and Associated Semantic Process for the Specification and Translation of Programming Languages. Ph.D. Dissertation. Department of Information and Computer Science, University of California, Irvine, 1974.

Hopwood, M.D. and G.L. Hopwood. "MOL620: A Machine Oriented Language and Language Compiler for the Varian Data 620/I." Department of Information and Computer Science, University of California, Irvine, Technical Report #1, September 1971 (revised November 1973).

Housel, B.C. A Study of Decompiling Machine Languages into High-Level Machine Independent Languages. Ph.D. Dissertation. Department of Computer Science, Purdue University, Technical Report TR 100, August 1973.

IBM. "1400 Autocoder to COBOL Conversion Aid Program" (360A-SE-19X), "Version 2 Application Description Manual" (GH20-0352-2), White Plains, New York: IBM 1967.

Ikezawa, M.A. and R.E. Kayfes. "A Structured Calculus for Program Analysis and Testing." Logicon, Inc., San Pedro, California, Technical Report No. CSS-75019, November 1975.

Ikezawa, M.A. Personal communications. January 1976. Los Angeles Chapter ACM SIGPLAN speech. October 1976.

Ikezawa, M.A. "AMPIC for the Non-Programmer." Logicon, Inc., San Pedro, California, Technical Report No. R:CSS-77004, May 1977.

Jensen, K. and N. Wirth. Pascal User Manual and Report. New York: Springer-Verlag, 1975.

Knuth, D.E. "Structured Programming with go to Statements." Computing Surveys, Vol. 6, No. 4, December 1974, pp. 261-301.

Knuth, D.E. The Art of Computer Programming, Vol. 1. Reading, Massachusetts: Addison-Wesley, 1968.

Loveman, D.B. "Program Improvement by Source-to Source Transformation." Journal of the Association for Computer Machinery, Vol. 24, No. 1, January 1977, pp. 121-145.

Lowry, E.S. and C.W. Medlock. "Object Code Optimization." Communications of the Association for Computing Machinery, Vol. 12, No. 1, January 1969, pp. 13-22.

Philippakis, A.S. "Programming Language Usage." Datamation, October 1973, pp. 109-114.

Philippakis, A.S. "A Popularity Contest for Languages." Datamation, Vol. 23, No. 12, December 1977, pp. 81, 86-87.

Rowe, L.A. A Formalization for Modelling Structures and the Generation of Efficient Implementation Structures. Ph.D. Dissertation. Department of Information and Computer Science, University of California, Irvine, 1976.

Sammet, J.E. Programming Languages: History and Fundamentals. Englewood Cliffs, New Jersey: Prentice-Hall, 1969.

Sammet, J.E. "Programming Languages: History and Future." Communications of the Association for Computing Machinery, Vol. 15, No. 7, July 1972, pp. 601-610.

Sassaman, W.A. "A Computer Program to Translate Machine Language Into Fortran." Proceedings of the Spring Joint Computer Conference, 1966, pp. 235-239.

Standish, T.A., D.C. Harriman, D.C. Kibler, and J.M. Neighbors. "Improving and Refining Programs by Program Manipulation." Proceedings of the 1976 Annual Conference of the ACM, October 20-22, 1976, pp. 509-516.

Steel, T.B., ed. Proceedings of the IFIP Working Conference on Formal Languages for Computer Programming. Amsterdam: North Holland, 1966.

Timmreck, E.M. "Computer Selection Methodology." Computing Surveys, Vol. 5, No. 4, December 1973, pp. 199-222.

Ultrasystems, Inc. Newport Beach, California. "Decompiler Program Development and High-Level Program Flow Charter Specification Development:"

"Decompiler Program Design Requirements," A002, document no. 74/6.29-2

"Decompiler Program Design Specification," A003, document no. 74/6.29-5

"Decompiler Program Design Document," A004, document no. 74/6.29-9

"Decompiler Test Plan/Test Report," A005, document no. 74/6.29-14.

Varian Data Machines. Varian Data 620/i Computer Manual. Irvine, California, 1968.

van der Poel, W.L. and L.A. Maarsen, eds. Machine Oriented Higher Level Languages, New York: American Elsevier Publishing Co., Inc, 1974.

Weissman, C. LISP 1.5 Primer. Belmont, California: Dickenson Publishing Co., Inc., 1967.

Appendix I

IMTEXT STATEMENTS

This appendix lists the IMTEXT statements implemented for the experiments conducted during this research. The statements are listed alphabetically by IMTEXT operator. The following meta symbols are used to indicate classes of items which may appear within a statement.

const	-- a symbolic or numeric constant;
exp	-- an IMTEXT statement appearing on the argument list of an enclosing statement by process of substitution, a variable name, or a constant;
ivar	-- indexed variable name;
label	-- a statement label;
nil	-- a placeholder for an unused item in the statement;
sop	-- a symbolic shift operator of the target language;
svar	-- synthetic variable name;
texp	-- a target designator; a variable name, or a substituted target expression (e.g., an INDEXED or INDIRECT expression);
tvar	-- a target variable name;

Each statement is described with its prototype format, a semantic description of its meaning followed by a target language statement and/or expression into which the IMTEXT statement might be translated. (MOL620 is used as the target language.) See Chapter 4 for an introduction to the concept and use of IMTEXT statements for decompilation.

(ADD, (exp1, exp2), texp)
    Add the value of exp1 to the value of exp2 and store the result in the target designated by texp.
    target statement: texp := exp1 + exp2;
    target expression: (exp1 + exp2)

(AND, (exp1, exp2), texp)
    Evaluate the boolean expressions exp1 and exp2. Form the boolean product of these values and store the result in the target designated by texp.
    target statement: texp := exp1 AND exp2;
    target expression: (exp1 AND exp2)

(B, texp, nil)
Branch to the designated location, texp. Texp may be a label, an indexed expression, or an indirect expression corresponding to simple, indexed, or indirect jumps.
target statement: GOTO texp;
target expression: none

(BAND, (exp1, exp2), texp)
Evaluate exp1 and exp2. Perform a bitwise AND of these values. Store the result in the location designated by texp.
target statement: texp := exp1 BAND exp2;
target expression: (exp1 BAND exp2)

(BOR, (exp1, exp2), texp)
Evaluate exp1 and exp2. Perform a bitwise OR of these values and store the result in the location designated by texp.
target statement: texp := exp1 BOR exp2;
target expression: (exp1 BOR exp2)

(BT, (exp, label), nil)
Branch on true condition. This IMTEXT statement always has three subgraphs attached to it in the control graph where it appears -- the "true" subgraph, the "false" subgraph, and the "join" subgraph (see Chapter 4). If the value of the expression is true, the true subgraph is executed, otherwise the false subgraph is executed. Following execution of the true or false subgraph, the join subgraph is executed. The label in the statement is merely annotation.

```
target statements: IF exp THEN ...;
          IF exp THEN ...  ELSE ...;
          WHILE exp DO ...;
target expressions: none
```

(BUMP, tvar, tvar)
The value of the location indicated by tvar is incremented by one. Tvar may not be indexed or indirect. (See BUMPI).
target statement: BUMP tvar;
target expression: none

(BUMPI, texp, texp)
This statement is a special case of the BUMP statement. Texp is an indirect or indexed designator. Increment the designated location by one.
target statement: BUMP texp;
target expressions: none

```
(BXOR, (exp1, exp2) texp)
     Evaluate exp1 and exp2.  Perform the bitwise
     exclusive-or of these values and store the result in
     the location designated by texp.
     target statement: texp := exp1 BXOR exp2;
     target expression: (exp1 BXOR exp2)

(CALL, texp1, texp2)
     Call procedure designated by texp1.  Return address is
     stored in location designated by texp2.
     target statement: CALL texp1; %texp2 is implicit%
     target expression: none

(CALLT, (exp, texp1), texp2)
     If value of exp is true, then call the procedure
     designated by texp1.  The return address is stored in
     the location designated by texp2.
     target statement: IF exp THEN CALL texp1;
     target expression: none

(DIV, (exp, texp1, texp2), (texp1, texp2))
     Divide the double length operand (texp1,texp2) by the
     value of the expression exp.  Store the double length
     result in (texp1, texp2).
     target statement: CALL DIV(exp);
               %texp1 and texp2 are implicit%
     target expression: none

(EQ, exp, texp)
     If the value of exp equals zero, then set the location
     designated by texp to true, else set texp to false.
     target statement: texp := (exp = 0);
     target expression: (exp = 0)

(EXC, (const1, const2), nil)
     Perform the external control I/O function const1 on
     device const2.
     target statement: CALL EXC (const1, const2);
     target expression: none

(GE, exp, texp)
     If the value of exp is greater than or equal to zero,
     then set the location designated by texp to true, else
     set texp to false.
     target statement: texp := (exp >= 0);
     target expression: (exp >= 0)
```

(HALT, const, nil)
    Halt the machine with indication const.
    target statement: ",HLT,const";
    target expression: none

(INDEX, (ivar, exp), svar)
    This statement is always substituted in a following IMTEXT statement either on the argument or change list. Svar is a synthetic variable designator used in the substitution. If this statement appears on an argument list, then it means that the value of the vector element (exp) in vector ivar is used. If the statement appears on the change list, then that vector element is modified by the execution of the enclosing statement.
    target assignment: ivar[exp] := ... ;
    target expression: ivar[exp]

(INDIRECT, exp, svar)
    Like the INDEX statement described above, this statement is always substituted in an argument list or a change list of another statement. Svar is the synthetic variable designator used in the substitution. If this statement appears on an argument list, then it means that the value of the location whose address is given by the value of exp is used. If the statement appears in the change list, it means the location whose address is the value of exp is modified by the execution of the enclosing statement.
    target assignment: [exp] := ...;
    target expression: [exp]

(INPUT, const, texp)
    Input data from device const and store the value in the location designated by texp.
    target statement: texp := INPUT(const);
    target expression: INPUT(const)

(JF, (exp, label), nil)
    If the value of exp is false then goto label.
    target statement: IF NOT exp THEN GOTO label;
    target expression: none

(LABEL, label, nil)
    Defines a symbolic label on the next IMTEXT statement.
    target statement: label:
    target expression: none

```
(LT, exp, texp)
     If the value of exp is less than zero, then set the
     location designated by texp to true, else set texp to
     false.
     target statement: texp := (exp < 0);
     target expression: (exp < 0)

(MOVE, exp, texp)
     This is the most basic IMTEXT statement.  The value of
     exp is stored in the target designated by texp.
     target statement: texp := exp;
     target expression: (exp)

(MOVEXP, exp, texp)
     Same as MOVE except the target is always explicitly
     assigned.  MOVEXP will always be substituted in the
     argument list of some other statement.  MOVEXP
     corresponds to an assignment expression in a target
     language.
     target statement: none
     target expression: (texp := exp)

(NOP, nil, nil)
     No operation.  Appears in timing loops or to allow I/O
     to complete.
     target statement: ",NOP,";
     target expression: none

(MUL, (exp, texp1, texp2), (texp1, texp2))
     Multiply the double length operand (texp1, texp2) by
     the value of exp.  Store the double length result in
     (texp1, texp2).
     target statement: CALL MUL(exp);
               %texp1 and texp2 are implicit%
     target expression: none

(OUTPUT, (exp, const), nil)
     Output the value of exp to device const.
     target statement: CALL OUTPUT(exp, const);
     target expression: none

(SHIFT, exp, texp)
     Perform the shifting instruction indicated by the value
     of exp.  The result affects texp.
     target statement: CALL SHIFT (exp);
               %texp1 is implied by exp%
     target expression: none
```

(SHIFTO, (exp, const, sop), texp)
    Special case of the shift statement used for the less complex forms of the source instruction. The value of exp is shifted using the symbolic operator, sop, by the shift count, const. The result of the shift is stored in texp.
    target statement: texp := exp "sop" const;
    target expression: (exp "sop" const)

(SENSE (const1, const2), svar)
    Always appears as an argument to a BT statement after forward substitution has taken place. Svar is a synthetic variable used in the forward substitution process. The value of a SENSE expression is true if status const1 is true for I/O device const2.
    target statement: none
    target expression: (SENSE(const1,const2))

(SUB, (exp1, exp2), texp)
    Subtract the value of exp2 from the value of exp1 and store the difference in the target designated by texp.
    target statement: texp := exp1 - exp2;
    target expression: (exp1 - exp2)

(XEC, label, nil)
    Execute the instruction at location label (one instruction subroutine). This IMTEXT statement is not supported in the target translation.
    target statement: XEC const;
            %will be flagged as an error when compiled%
    target expression: none

(XIF, (exp, label), nil)
    Execute the instruction at location label if the value of exp is true. Like XEC above, this statement is unsupported in the target language.
    target statement: IF exp THEN XEC label;
            %will be flagged as an error when compiled%
    target expression: none

Appendix II

SUMMARY OF 620/i INSTRUCTION SET

As explained in Chapter 5, the source machine/language chosen for the decompilation experiments of this research was the Varian Data Machines 620/i minicomputer. This appendix describes the instruction set and relevant features of that machine.

The 620/i was first delivered in 1968. It was a 16-bit minicomputer implemented using TTL circuitry and based on an earlier discrete component machine called the 620/A, one of the first 16-bit minis. The 620/i has since been succeeded by an upward compatible family of machines -- the 620/L, 620/F, and most recently the V70 series of computers (1972-1977).

The 620/i has three programmable registers:
A-register: the general purpose 16-bit accumulator; may not be used for indexing;
B-register: accumulator extension and index register;
X-register: index register.

The memory is word addressable to 32K locations. The high order bit of an address word usually specifies indirection. The original memory of the machine was implemented in 1.8 microsecond synchronous core memory in 4K modules.

The instruction set is not symmetric in the sense that the B and X registers are not general purpose registers and only a partial set of condition testing/branching instructions are implemented. There are five common addressing modes for most functional instructions involving the registers and memory:

direct -- single word instructions may directly address the first 2K of memory; double-word instructions may reference the entire 32K.

indirect -- one word instructions may address the first 512 words of memory; double-word instructions may reference the entire memory; multi-level indirection is permitted to any level.

relative -- single word addressing relative to the location counter up to 511 words forward; double-word instructions may address the whole memory space.

indexed -- single word addressing relative to the contents of the B or X registers with a displacement of 0-511 words; double-word instructions include a full word of displacement.

immediate -- the operand is in the second word of the instruction.

Program counter modification instructions (branch, subroutine call, and execute) may be executed based upon the contents of the registers. Nine simultaneous conditions may be tested at once. (See Figure 4-F.) These instructions may reference the target directly or indirectly over the full addressing range. All of these instructions are two words long.

The only instruction provided for subroutine calling in the 620/i is the "Jump and Mark" instruction. The return address is stored in the first word of the subroutine and execution begins at the second word of the subroutine. A return is effected with an indirect jump through the "mark word." (Of course, reentrant programming is impossible to do with any efficiency on this machine due to the absence of a subroutine calling instruction which does not modify memory.)

Following is an alphabetical list of some of the more commonly used instruction mnemonics of the 620/i and their meanings:

ADD -- Add memory to A-register.
ANA -- AND memory with A-register.
AOFA,AOFB,AOFX -- Add overflow register to A, B, or X-register.
ASLA, ASLB -- Shift A or B left arithmetically 0-31 bit positions.
ASRA,ASRB -- Shift A or B right arithmetically 0-31 bit positions.
CIA, CIAB, CIB -- Input data word from device to cleared register.
CPA, CPB, CPX -- One´s complement register contents.
DAR, DBR, DXR -- Decrement register by one.
DIV -- Divide (A, B) pair by memory word. Remainder to A, quotient to B.
ERA -- Exclusive-OR A-register with memory.

EXC -- I/O device external control function.
HLT -- Halt the cpu.
IAR, IBR, IXR -- Increment the register by one.
INA, INAB, INB -- Inclusive-OR I/O data word to register.
INR -- Increment a memory location by one.
JMP -- Conditional and unconditional jump (see Figure 4-F).
JMPM -- "Jump and Mark" subroutine call. Same conditions as JMP.
LASL, LASR -- Long arithmetic shift (left and right) of (A, B) pair, 0-31 bit positions.
LDA, LDB, LDX -- Load register from memory.
MUL -- Multiply the (A, B) pair by a memory word. Double word product to (A, B).
NOP -- No operation.
OAB, OAR, OBR -- Output word of data from register to I/O device.
ORA -- Inclusive-OR memory word with A-register.
ROF, SOF -- Reset or set overflow flag.
SEN -- Sense status of I/O device.
SOFA, SOFB, SOFX -- Subtract overflow register from A, B, or X.
STA, STB, STX -- Store register to memory.
SUB -- Subtract memory from A-register.
TAB, TAX, TBA, ... -- Transfer register value to another register.
XEC -- Execute a remote instruction. Same conditions as for JMP.
ZERO -- Zero a register.

Appendix III

SUMMARY OF THE MOL620 LANGUAGE

The MOL620 language was designed in 1969 and must be considered obsolete by current programming language standards, even in the context of its intended purpose, i.e., a machine-oriented replacement for assembler language. Many missing features, such as variable type definition (ala Pascal) and lack of local environments reflect the inadequacies of the early versions of the metacompiler used to implement the translator. However, a large amount of reliable software was successfully written using MOL620. The code generated by the compiler has been efficient enough in time and space to keep programmers from slipping back into assembler language. In addition, because of the constant turnover of students in the University environment, the improved readability of the MOL620 programs versus assembler language has been very important.

A superset language, MOLSUE (Hopwood 1976), was very successfully used for five years by programmers on the Distributed Computing System Project (Farber 1973) at UCI. Many of the inadequacies of MOL620 were corrected in the design and implementation of MOLSUE. The similarities between the two languages allowed programmers to use knowledge of either language to easily write programs for both the Varian 620/i and the Lockheed Sue, even though the machines have very different architectures and native instruction sets.

The following part of this appendix is an excerpt taken from the report (Hopwood and Hopwood 1971) describing the MOL620 language.

EXPRESSION EVALUATION

Expression evaluation consists of combining operand values according to the rules given by the operators in the expression. The sequence of the application of the operators to the operands is given by the hierarchy of the operators, grouping of sub-expressions by parenthesization, and normal left-right sequencing.

Operands

Operands are symbols or expressions. The basic operands are: identifiers, quoted names, numbers, and quoted characters.

identifiers -- sequence of one to six alphanumeric (A-Z,0-9) characters beginning with an alphabetic character e.g. A, Q32, ICS, NAMELY

quoted names -- any string of characters enclosed in double quotes. The string in quotes should be a meaningful symbol to the 620/i DAS assembler. E.g., "CB", "GAMMA+5", "A-2". These names should not be used in the normal course of creating a MOL620 program.

numbers -- decimal, octal, and hexadecimal

decimal numbers -- sequence of decimal digits optionally preceded by a + or - sign. Sixteen bit limit implies decimal number of -32768 to +32767.

octal numbers -- sequence of octal digits preceded by a zero. Range of 000000 through 0177777.

hexadecimal numbers -- one to four hex digits (0-9,A-F) preceded by H´ and followed by ´. For example, H´FFFF´, H´ABC´, H´E0´, H´1234´.

quoted characters -- one or two ASCII characters enclosed in single quotes, e.g., ´A´, ´AB´. Characters are coded as eight bit quantities with the high order bit on. A single character is right justified in a word of zeroes.

Special Names

There are several implementation dependent names which the programmer may need to access on occasion

(AR) accumulator, A-register
(BR) auxiliary accumulator and index reg, B-register
(XR) index register, X-register.

Function Invocation

A function procedure is called with zero to three arguments passed by value. The value of the function is the value

RETURNed by the procedure. (See RETURN statement.) The parentheses after the function name are required in this context.

Examples,
```
        F()
        F(A+B)
        ALPHA(CHAR)
        G(X, @W, Z+3)
```

The arguments are expressions. They are evaluated in an undefined order. The evaluated argument values are passed to the function in the A, B, and X registers, first argument in the A-register, second in the B-register, third in the X-register.

Array and Offset Referencing

An array or offset reference is of the form: name [exp], where name is an identifier and exp evaluates to an index if the name specifies an array, or a pointer if the name specifies an offset.

Examples,
```
        TABLE[I]
        MEM[PC()+Q-5]
```

Indirection

An indirect expression has the form: [exp]. The value of the indirect expression is the word value of the memory location whose address is the value of the exp. For example, the following two expressions are exactly equivalent in that they both fetch (or store) a value from the same physical memory location: A, [@A].

Assignment Expressions

Another type of operand in an expression is the assignment expression. (See "Assignment Statement".) The assignment expression has the form

(1) target name
(2) ':=', the assignment operator
(3) expression

The expression is evaluated and assigned to the target. The value of the assignment expression is the value of the assigned target.

Examples,
```
        A+(B:=C-D)
        F(A:=X, 0)
        A:= B := C := D ;
```

In the first example, B is assigned (C-D) then the sum of A and the value (C-D) is computed. In the second example, the evaluation of the first argument to the function F causes A to be assigned the value of X. In the last case A, B, and C are assigned the same value, namely, that of D.

IF Expression

An IF expression has the form: IF exp1 THEN exp2 ELSE exp3.

If exp1 is true then the value of the IF expression is exp2 else the value is exp3. An IF expression may be used any place an expression may appear.

Examples,
```
        IF A THEN B ELSE C
        IF B=1 THEN IF C=2 THEN 3 ELSE 1 ELSE 0
        A := IF F(B) THEN C+D ELSE C-D ;
        ABSA := IF A>=0 THEN A ELSE -A ;
        MINAB := IF A<=B THEN A ELSE B ;
```

Note that unlike the IF statement, the IF expression requires an ELSE clause.

The last three examples are assignment statements using an IF expression to calculate the assignment value.

Operators

The only data type of MOL620 is a sixteen bit binary value. No coercion of types is performed since there is only one type. The operators can be classified into six groups:

(1) arithmetic	--	two's complement arithmetic
(2) relational	--	comparing arithmetic quantities
(3) logical	--	combining truth values
(4) bitwise	--	masking, inserting bits
(5) shifting	--	logical and arithmetic shifts
(6) address	--	get the physical address of a name

The truth values are defined as:

```
        true -- any non-zero value
        false -- the zero value.
```

Arithmetic Operators

+ exp	unary plus
- exp	unary minus
exp1 + exp2	summation
exp1 - exp2	difference
exp1 * exp2	multiplication (one word result)
exp1 / exp2	division (quotient; remainder in BR)

Relational Operators (arithmetic compare)

exp1 < exp2	true iff exp1 is less than exp2
exp1 <= exp2	true iff exp1 is less than or equal exp2
exp1 = exp2	true iff exp1 is equal exp2
exp1 # exp2	true iff exp1 is not equal exp2
exp1 >= exp2	true iff exp1 is greater than or equal exp2
exp1 > exp2	true iff exp1 is greater than exp2

NOTE -- if the absolute value of (exp1-exp2) is greater than 32,767 the results of the relational operations (<,<=,>=,>) are not valid in all cases due to overflow. If your data might be in this range your program will have to test for overflow or carry explicitly in assembly language.

Logical Operators

NOT exp	true iff exp is false
exp1 AND exp2	true iff exp1 is true and exp2 is true
exp1 OR exp2	true iff exp1 is true or exp2 is true (inclusive-OR)
exp1 XOR exp2	true iff one (but not both) of the operands are true (exclusive-OR)

Bitwise Operators

exp1 BAND exp2	bitwise AND
exp1 BOR exp2	bitwise inclusive-OR
exp1 BXOR exp2	bitwise exclusive-OR

The bitwise operators operate on all sixteen bits of the operands in parallel. They are implemented by the machine instructions ANA, ORA, and ERA. BAND is useful for masking operations. BOR is used to insert bits in a word. BXOR is useful for complementing bits of a word.

Shift Operators

exp1 "LSRA" exp2	shift exp1 right logical exp2 places; insert zeroes on left
exp1 "ASLA" exp2	shift exp1 left arithmetic exp2 places; insert zeroes on the right
exp1 "ASRA" exp2	shift exp1 right arithmetic exp2 places; insert zeroes on the left
exp1 "LRLA" exp2	rotate exp1 left logical exp2 places;

The A/B long shifts LASL, LLRL, LASR, and LLSR are also permitted.

Address Operator

@ symbolic constant	assembly time value of the constant
@ identifier	physical address of identifier
@ identifier [exp]	physical address of array element

Assignment Operator

See "Assignment Expression".

Precedence of Operators

The precedence of operators is given in the following table. When two operators seem to share the same operand the operator with the highest binding will operate on the operand.

```
unary +, -, @        highest binding
shift operators
BAND
BOR, BXOR
*, /
<, <=, =, #, >=, >
NOT
AND
OR, XOR
:=                   lowest binding
```

Operators at the same level are performed left to right. Evaluation of the operands of an operator (except AND and OR) may be done in either order by the compiler.

The logical operators, AND and OR, evaluate their operands left to right. If the evaluation of the left hand operand strictly determines the result of the operation, then the right hand operand is not evaluated.

Examples,

A AND B	A is evaluated. If false, the result must be false and B is not evaluated, else the result is the value of B.
A OR B	A is evaluated. If true, the result must be true and B is not evaluated, else the result is the value of B.

Of course, parentheses may be used in an expression to group operands with their respective operators.

PROGRAM

The top level syntactic entity which is accepted by the MOL620 compiler is called a program. A program consists of a set of compiler directives, global declarations (DECLARE, EQU, SET, INTERNAL, and EXTERNAL statements), DAS statements, and procedures.

<u>DONE</u> <u>statement</u>

A program is terminated with a DONE statement. This has two forms:

(1) DONE.
(2) DONE <u>start</u>.

where <u>start</u> is the name of the first (main) procedure to be invoked when the program is loaded. If the main procedure is not in the compilation file, the first alternative of the DONE statement should be used, i.e., no start name.

Semicolons

In MOL620 the semicolon (;) is used as a terminator rather than a separator as in Algol. A semi-colon is used to terminate every statement in the language (except the DONE statement) in the same way a period is used in English. Two or more semicolons never appear next to each other. One is enough to terminate the top level of a nested statement. The statement before an END keyword in a block always is terminated with semicolon, unlike Algol.

PROGRAM COMPONENTS

Compiler Directives

Compiler directives are used to inform the compiler about various user desired compile time options. Compiler directives may appear anywhere a DAS statement may appear.

Listing Control -- .LIST, .NOLIST

The .LIST directive causes following MOL620 source code (including comments) to be inserted in the assembly language output of the compiler as comments.

.NOLIST turns off the listing option.

Tracing Control -- .TRACE, .NOTRACE

The .TRACE directive causes a " JMPM TRACE " instruction to be generated at the beginning of each procedure compiled while the trace directive is in effect. The TRACE procedure is assumed to be supplied by the user.

.NOTRACE turns off the trace option.

Literal Constant Generation Control -- .LITERAL, .NOLITERAL

.LITERAL directs the compiler to generate instructions using the literal pool built by the DAS assembler. The statement

```
F := 10;
```

would compile as

```
                      LDA =10
                       STA F.
```

.NOLITERAL directs the compiler to use double word immediate instructions. The statement

```
                      F := 10;
```

would compile as

```
                      LDAI 10
                       STA F.
```

Indirect Reference Control -- .INDIRECT=0, 1, or 2

.INDIRECT=0 directs the compiler to generate conditional assembly statements yielding a single word instruction if the indirect word is located in the first 512 locations of memory. Otherwise, a double word instruction is generated.

.INDIRECT=1 directs the compiler to generate single word indirect instructions.

.INDIRECT=2 directs the compiler to generate double word indirect instructions.

Indexed Reference Control -- .INDEXED=0, 1, or 2

.INDEXED=0 directs the compiler to generate conditional assembly statements yielding a single word instruction if the index displacement is an absolute value in the range 0..2047. Otherwise, a double word instruction is generated.

INDEXED=1 directs the compiler to generate single word indexed instructions.

INDEXED=2 directs the compiler to generate double word indexed instructions.

Note--.INDEXED=0 and .INDIRECT=0 imply that the symbol involved in the conditional assembly time testing is absolute and defined before it is used, otherwise assembly phase errors may result.

Relocatability Control -- .ABSOLUTE, .RELOCATE

.ABSOLUTE tells the compiler that the code produced will be assembled as an absolute binary image (not relocatable).

.RELOCATE tells the compiler that the program is to be relocated after assembly by a linking loader. This mode is only valid for use with the UCI DASMR assembler. To acheive standard Varian DASMR compatible code, use the following directives: .ABSOLUTE, .INDIRECT=2, .INDEXED=2.

Comments

Comments are enclosed in percent signs (%). A comment may appear anywhere in the source text. This implies a percent sign may be used only as a comment delimiter.

Examples,

```
        % THIS IS A COMMENT %
        DECLARE  A,          % SO IS THIS %
                 B,         % AND THIS %
                            % AND THIS ALSO %
                 C;   % AND THIS TOO %
        IF A = 0 THEN       % A COMMENT CAN GO HERE %
             CALL SUBR (A,     % OR HERE %
                        B,     % OR HERE %
                        C )    % OR HERE %;
```

DAS statement

A DAS statement is a string of characters enclosed in double quotes ("). The string is copied to the output file without quotes. The DAS statement is used to insert assembly language statements into a MOL620 program.

Examples,

```
        " SEN 0101,TTYOUT";     % sense teletype output ready %
        " NOP";                % a no-operation instruction %
```

SYMBOL DEFINITIONS AND VARIABLE DECLARATIONS

All symbols defined in a MOL620 program, including the "formal" parameters of procedures, are global to the entire program. There are no local symbols in MOL620.

INTERNAL statement

The INTERNAL statement is used to define names to the system linking loader . INTERNAL names are those defined in the current compilation file but are needed by a program in another load module.

The INTERNAL statement consists of two parts:

(1) the keyword INTERNAL
(2) name list -- one or more names separated by commas

Example,

```
        INTERNAL A, B, C;
```

EXTERNAL statement

The EXTERNAL statement is used to define names to the system linking loader. EXTERNAL names are those referenced in the current compilation file but are defined in a separate compilation file.

The EXTERNAL statement consists of two parts:

(1) the keyword EXTERNAL
(2) name list -- one or more names separated by commas

Example,

```
        EXTERNAL A, B, C ;
```

EQU and SET statements

The EQU and SET statements are used to define symbols in terms of other symbols which have previously been defined. This statement is used in many ways to define values for symbols at assembly time of the MOL620 program.

The EQU statement differs from the SET statement only in the sense that EQU defines symbols which have not been defined previously. SET redefines a symbol.

The EQU and SET statements consist of two parts:

(1) the keyword EQU or SET
(2) name-value list, one or more items separated by commas
    a. name
    b. ´=´
    c. name or constant (previously defined)

Examples,

```
        EQU A=0, B=1 ;
        EQU TRUE=1, FALSE=0, YES=TRUE, NO=FALSE ;
```

DECLARE statement

The DECLARE statement is used to define symbols as the names of memory locations. These symbols are variable names. The variables may be assigned an initial value in the declaration. At present, arrays of only one dimension may be declared.

The DECLARE statement consists of two main parts:

```
        (1) the keyword DECLARE
        (2) declared item list --
            a.  identifier
            b.  dimension (optional)
                a.   [constant] (array size)
             or
                a´.  [constant  :  constant]  (low  and high
                     indices)
            c.  initial value list (optional)
                a.  ´=´
                b.  constant
             or
                b´.  list of constants
                    a.  ´(´
                    b.  one  or  more constants separated by
                        commas
                    c.  ´)´
```

Examples,

```
        DECLARE A, B, C, D[5], E[0:10], F[-5:5] ;
        DECLARE
          TITLE = (19,´MOL620 VERSION 7.00´),
          MASK = 0177400, MAX = 99;
```

In the above examples A, B, and C are word scalars. D is word array of five elements, singly dimensioned. D should be referenced with indices zero through four. E is a vector of eleven words indexed from zero through ten. F consists of eleven words F[-5], ...,F[0], ...,F[5].

In the second example, TITLE is a vector long enough to hold its initial values. The value 19 is stored in one word, then the character string is packed two characters per word into the next ten words.

Declarations of variables may be placed anywhere in the program. Because of the 620/i relative addressing mode, if the declarations are placed after their references (but within 512 words), then single word instructions can be used to reference them. The MOL620 compiler assumes all scalars can be referenced in a single word and generates single word instruction mnemonics.

Procedures

A procedure is a closed subroutine or function. A procedure may not contain another procedure. A procedure consists of three parts:

(1) head
(2) body
(3) close.

The procedure head consists of the word PROC (or PROCEDURE) followed by an identifier which serves as the procedure name, followed by an optional list of 0 to 3 identifiers enclosed in parentheses.

Examples of procedure heads,

```
        PROC ALPHA(X);
        PROCEDURE F(Q1,R,V);
        PROC SUB1;
        PROCEDURE SUB1();
        PROC G3(X,,Z);
```

If a parameter list appears in the procedure heading, then on entry to the procedure the A-register is stored in the location specified by the first parameter, the B-register in the second, the X-register in the third. The parameters are not dummies in the sense that they are not local to the procedure but have scope over the entire DAS program generated.

The body of a procedure is empty or consists of a sequence of statements, comments, and declarations.

A procedure is closed by the word ENDP;. This has the same effect as the RETURN statement with no arguments.

Examples,

```
        PROCEDURE ADD (ARG1,ARG2);
        RETURN (ARG1+ARG2);
        ENDP;
        DECLARE ARG1,ARG2;

        PROCEDURE LINEARSEARCH (FIRST,N,VALUE);
        %THE ARGUMENTS ARE:
         FIRST = ADDRESS OF VECTOR TO BE SEARCHED
         N     = SIZE OF ARRAY (1 TO N)
         VALUE = VALUE TO BE FOUND.
         WE USE A LINEAR SEARCH TECHINIQUE TO RETURN
         THE INDEX OF THE VECTOR ELEMENT WHICH
```

```
 CONTAINS THE VALUE. (ASSUME IT IS THERE)%
LAST:=FIRST+N-1; %ADDR OF LAST WORD IN VECTOR%
FOR I:=FIRST UNTIL LAST DO
 IF [I]=VALUE THEN RETURN (I-FIRST+1);
ENDP;
DECLARE FIRST,LAST,I,N,VALUE;
```

A procedure should only be entered via a function call or subroutine call, and not by falling into it from preceding code.

PROCEDURE COMPONENTS

Assignment statement

An assignment statement consists of three parts:

(1) one or more target symbols separated by commas
(2) ':=', the assignment operator
(3) an expression.

A target symbol for an assignment is an identifier followed optionally by a bracketed index, or an indirect target -- an expression in square brackets:

identifier (letters and numbers starting with a letter)
identifier[exp] (an indexed name)
[exp] (an indirect target)

The assigned expression is evaluated and stored to the target locations right to left on the target list.

Examples,

```
A := B ;
C, D, E := ALPHA / BETA ;
A := B := C := 0 ;     %same as A,B,C := 0 %
REACT := A OR B AND NOT C ;
LINK[X], X := LINK[Y] ;
[Y+Z] := (N-2) "LSRA" 2 ;
```

The assignment

```
LINK[X] , X := LINK[Y] ;
```

is equivalent to

```
T := LINK[Y] ; X := T ; LINK[X] := T ;
```

for some temporary variable T.

With the target form [exp], the expression in brackets is evaluated, the resultant value is the address where to store the assigned value. (620/i multi-level indirection will occur if the sign bit of the indirect word is on.)

BUMP STATEMENT

The BUMP statement provides a convenient way to increment (by one) the value of a memory location. It consists of two parts:

(1) the keyword BUMP
(2) a list of one or more target names separated by commas.

Examples,

```
        BUMP I ;
        BUMP A, B, C[I-5] ;
```

The targets are incremented left to right on the target list.

IF statement

Conditional execution of a statement or block of statements can be accomplished through use of the IF statement. The IF statement consists of four parts:

(1) the keyword IF
(2) a conditional expression
(3) THEN part
    a. the keyword THEN
    b. statement
(4) ELSE part (optional)
    a. the keyword ELSE
    b. statement

Examples,

```
        IF A=B THEN H:=1 ELSE H:=2 ;
        IF F(N) THEN CALL SUB1(N) ;
        IF ALPHA THEN BEGIN A:=1 ; GR:=-77 ; END ;
        IF FLAG#0 THEN RETURN ;
        IF A=B THEN BEGIN V:=0 ; A:=B-1 ; END ;
                ELSE A:=B;
```

The expression following the keyword IF is evaluated to a boolean result . If the result is true, the THEN part is executed. If the result is false and there is no ELSE part, the following statement is executed. If the IF expression is false and there is an ELSE part, that is executed.

ELSE clauses in nested IF statements are associated with the closest preceding IF, (at the same syntactic level), that has no ELSE associated with it. The NULL statement is useful in providing an empty ELSE clause for an IF statement.

Example,
```
        IF A THEN IF B THEN C:=1 ELSE D:=1
```
means
```
        IF A THEN
          BEGIN
            IF B THEN C:=1 ELSE D:=1;
          END
```

FOR statement

A FOR statement is used for iteration across a statement or group of statements. The FOR statement consists of five parts:

```
        (1) the keyword FOR
        (2) the assignment part
            a.  variable name
            b.  ´:=´
            c.  expression
        (3) STEP part (optional)
            a.  the keyword STEP
            b.  expression
        (4) termination part
            a.   the keyword TO
            b.   expression
         or
            a´.  the keyword WHILE
            b´.  expression
        (5) DO part
            a.  the keyword DO
            b.  statement
```

The variable name target in the assignment part is called the index variable. It is initialized, and the test of the termination part is executed, i.e., the test is always executed <u>before</u> the DO part is performed.

If the test of the termination part is a TO form, then the value of the index variable is compared with the TO expression. If the index variable is less than or equal to the value of the expression, (greater than or equal if the STEP expression is negative), then the DO part is executed. Otherwise, control passes to the statement after the FOR statement.

If the test of the termination part is a WHILE form, the value of the WHILE expression is computed. If the value of the expression is true, the DO part is executed. Otherwise, control passes to the statement after the FOR statement.

Whenever the DO part has been executed, the index variable is incremented by the value of the STEP expression. If the STEP expression is missing, the index variable is incremented by one. After the index variable is incremented, control passes to the TO or WHILE test, and the loop begins again.

Examples,

```
        FOR I:=1 TO 10 DO A[I] := I*I ;
        FOR I:=N STEP -1 TO 1 DO A[I] := A[I]/(I*I) ;
        FOR J:=K-3 STEP M+J WHILE V<R DO V=V+J ;
```

The first statement will set A[I]=I*I FOR 1<=I<=10. The second is an example of counting the index variable down. The third uses a WHILE test to terminate the loop.

It is important to note that the expression of the STEP part, and termination part as well as any expression involved in calculating the effective address of the index variable are evaluated each time through the loop. In the third example M+J will be evaluated each time the STEP value is needed. If such expressions are constant throughout the execution of the FOR statement they should be assigned to a simple variable for the sake of efficiency.

REPEAT statement

This statement provides a means of performing a statement as long as some condition is false. The condition is tested after the statement is executed. The REPEAT statement consists of four parts:

(1) the keyword REPEAT
(2) a statement
(3) the keyword UNTIL
(4) an expression.

Examples,

```
        REPEAT A[I] := 0 UNTIL I > @MAX ;
        REPEAT X := LINK[X] UNTIL X = 0 ;
```

The REPEAT statement body is executed. The UNTIL expression is then evaluated. If the result is false then the body is executed again until the condition is true.

This statement is similar to the WHILE statement except the body is always performed at least once, since the test for termination is done after the statement is executed.

WHILE statement

This statement provides a means of performing a statement as long as some condition is true. The condition is tested first. The WHILE statement consists of three parts:

```
        (1) the keyword WHILE
        (2) a conditional expression
        (3) DO part
            a.  the keyword DO
            b.  statement
```

Examples,

```
        WHILE A < B DO CALL SUB1(@A, @B) ;
        WHILE I <= N DO BEGIN A[I] = I ; BUMP I ; END ;
        WHILE X # 0 DO X := LINK X ;
```

The WHILE condition is evaluated. If the result is true then the DO part is executed. The test and execution are repeated until the WHILE expression is false. Control then passes to the following statement.

CALL statement

The CALL statement is used to invoke a procedure. It consists of three parts:

```
        (1) the keyword CALL
        (2) a procedure name, or indirect target
        (3) argument list (optional)
            a.  '('
            b.  list of zero to three  expressions separated
                by commas (optional)
            c.  ')'
```

Examples,

```
        CALL SUB1 ;
        CALL ABC (X, Y) ;
        CALL F (@OP, @RESULT, D[X]+1) ;
        CALL [PTAB[I]] (CHAR) ;
```

The arguments are evaluated in an undefined order and passed by value to the called procedure. This first argument value is passed in the A-register, the second in the B-register, and the third in the X-register. CALLing a procedure is equivalent to invoking the procedure as a function except the RETURNed value (if any) is discarded.

In the last example PTAB is a table of procedure names indexed by I. PTAB[I] is evaluated and the result is the address of the target procedure. CHAR is passed by value to this target procedure.

RETURN statement

Exit from a procedure, optionally returning an expression value, is done with the RETURN statement. The RETURN statement consists of two parts:

```
        (1) the keyword RETURN
        (2) an optional argument
            a.  '('
            b.  expression (optional)
            c.  ')'
```

Examples,

```
        RETURN ;
        RETURN () ; %same as previous stmt%
        RETURN (EXPR) ;
```

If an argument is specified, it is evaluated and returned to the invoking procedure in the A-register. If the procedure was invoked as a function, then the value will be used, otherwise it will be discarded.

NULL statement

The NULL statement creates no machine code. It may be used as a dummy statement in an IF..THEN...ELSE construct or as something to which a label can be attached.

Examples,

```
        LABEL: NULL ;
        IF A=5 THEN
                IF B=3 THEN CALL SUB3
                ELSE NULL
        ELSE CALL SUB5 ;
```

GOTO statement and labels

A label may be associated with a statement by preceding the statement with an identifier and a colon. Unconditional transfer of control to a labelled statement is done with the GOTO statement. The target of a GOTO may also be specified with indirection.

Examples,
```
        L: GOTO L ;        %infinite loop%
        GO TO LOOP ;
        GOTO [J] ;
        GO TO [TABLE[I]] ; %thru branch table%
```

In the form, GOTO[exp] the expression is evaluated and the result is the address of the jump target.

The use of the GOTO statement in MOL620 must be carefully considered. Normally, all targets of a GOTO should be in the same procedure with the GOTO statement.

STOP statement

The STOP statement is used to cause the machine to halt at execution time.

Examples,
```
        STOP ;
        HALT ;
```

Obviously, in a time-sharing program this statement should not appear.

Compound Statement (Block)

A block is a group of statements preceded by the word BEGIN and followed by the word END. A block is used to group statements logically together for the IF, WHILE, FOR, and REPEAT statements. Blocks may be nested to any level.

Examples,
```
        BEGIN A:=1; B:=2; C:=3; END;
        FOR I:=1 TO N DO
          BEGIN A[I]:=I; B[I]:=0; END;
        IF X=0 THEN BEGIN Z:=F(B, C); X:=1; END;
```

Miscellany .

DAS statement and declarations may also appear inside a procedure . Compiler directives in a procedure are allowed. Normally, of these top level statements, only DAS statements are written inside the scope of a procedure.

CODE GENERATION

Certain details of the code generation of the compiler are useful to understand during the debugging phase of program testing.

Expression evaluation -- all expressions are evaluated in the A-register, unless the value may be directly loaded into its target register (different than A). If temporary variables are needed during the expression evaluation, space for them is set asside at the end of the procedure.

Subroutine linkage -- subroutines are called with the DAS instruction " JMPM SUBR".

Argument passing -- is handled the same way for functions and subroutine CALLs. The arguments are evaluated and placed in the A, B, and X registers, corresponding to the first, second, and third argument. The limit on the number of arguments is three.

Procedure prologue -- when a procedure is entered, after the JMPM instruction is executed, the registers are stored in the variables appearing in the "formal" parameter list of the called procedure.

The return address is stored by the JMPM instruction at the first location of the procedure.

Procedure epilogue -- the argument (if any) of the RETURN statement is evaluated into the A-register. An indirect jump through the "mark word" of the procedure is executed. If execution of a procedure reaches the ENDP statement, a RETURN is simulated.

Appendix IV

EXCERPT OF THE ISADORA SOURCE PROGRAM

This appendix contains the assembler listing output text of the ISADORA 620/i program used as a test case in the decompilation experiment described in Chapter 6. The first 1226 lines of source listing (out of 2041) are reproduced here in a compressed format to reduce the volume of this appendix.

See Appendix II or a 620/i Reference Manual (Varian 1968) for a description of the op code meanings.

See Appendix V for the decompiled text created from this source data.

```
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 1

        *
        *       ISADORA DEBUGGER -- ICS DEPT -- UC IRVINE
        *       VERSION 2.20
        *       ORIGINAL AUTHOR: MICHAEL PEPPER
        *                        ICS DEPT, UCI, 1969
        *
        *       LAST UPDATE:     G.L. HOPWOOD
        *                        10-JUL-75
        *
033000  BGNG    ,EQU    ,033000
000200  LO      ,EQU    ,0200
        *LO     ,SET    ,END
        *HI     ,EQU    ,037777 UPPER BOUND OF USER AREA
032777  HI      ,EQU    ,BGNG-1 UPPER BOUND OF USER AREA
077630  BLD1    ,EQU    ,077630 ADDR OF START OF LOAD ROUTINE
077434  BLD2    ,EQU    ,077434 ADDR OF START OF DUMP ROUTINE
077540  MON     ,EQU    ,077520+16
000000  PIM     ,EQU    ,0
000001  PMLA    ,EQU    ,1
000006  PMLB    ,EQU    ,6
000375  MSKA    ,EQU    ,0375
000277  MSKB    ,EQU    ,0277
000004  IDVA    ,EQU    ,4      SERIAL CONTROLLER INPUT IF SS3 I
000001  IDVB    ,EQU    ,1      SERIAL CONTROLLER INPUT IF SS3 I
000004  ODVA    ,EQU    ,4      SERIAL CONTROLLER OUTOUT IF SS3
000001  ODVB    ,EQU    ,1      SERIAL CONTROLLER OUTPUT IF SS3
000000  ODV     ,EQU    ,0      SERIAL CONTROLLER OUTPUT FOR 611
        *USER SHOULD SPECIFY THE FOLLOWING PARAMETERS BEFORE ASS
        *    BGNG   ORIGIN POINT FOR ISADORA
        *    PIM    PIM GROUP NUMBER (0,1,2)
        *    PMLA   INPUT PIM LINE FOR INPUT DEVICE IF SS3 IS OFF
        *    PMLB   INPUT PIM LINE FOR INPUT DEVICE IF SS3 IS ON.
        *    MSKA   PIM MASK IF SS3 IS OFF.
        *    MSKB   PIM MASK IF SS3 IS ON.
        *    IDVA    INPUT DEVICE NUMBER IF SS3 IS OFF.
        *    IDVB    INPUT DEVICE NUMBER IF SS3 IS ON.
        *    ODVA    OUTPUT DEVICE NUMBER IF SS3 IS OFF.
        *    ODVB    OUTPUT DEVICE NUMBER IF SS3 IS ON.
        *    C611   ADDRESS OF 611 CHARACTER OUTPUT ROUTINE
        *           IF HE IS OUTPUTTING TO TEK SCOPE
        *    LO     LOWER BOUND OF USER WORK AREA
        *    HI     UPPER BOUND OF USER WORK AREA
        *    BLD1   LOCATION OF START OF LOAD ROUTINE
        *    BLD2   LOCATION OF START OF DUMP ROUTINE
        *
        *DEBUGGING PACKAGE - INITIALIZATION ROUTINE
        *ROUTINE ENABLES AND OUTPUTS MASKS FOR ALL PIMS
        *
        *THERE ARE THREE INTERRUPT SERVICE ROUTINES:
        *       INA -- INTERRUPTS WHILE RUNNING DEBUGGER COME HE
        *       INP -- INTERRUPTS WHILE RUNNING USER PGM "
        *       ANP3 - INTERRUPTS WHILE RUNNING ALL-NOT FUNCTION
        *
033000          ,ORG    ,BGNG   ORG AT SPECIFIED LOC
000060  PMIA    ,SET    ,020*PIM+060
000062  PMIA    ,SET    ,2*PMLA+PMIA    PIM INT LOCATION IF SS3
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 2

000060  PMIB    ,SET    ,020*PIM+060
000074  PMIB    ,SET    ,2*PMLB+PMIB    PIM INT LOCATION IF SS3
        *
```

```
                     *
                     *
033000 001000  START    ,JMP     ,INIT
033001 033004
033002 131256  VERSN    ,DATA    ,'2.20' IN CORE VERSION NUMBER
033003 131260
       033004  INIT     ,EQU     ,*
033004 005301           ,DECR    ,1      A=-1
033005 001400           ,JSS3    ,DFLT   INITIALIZE TTY
033006 033010
033007 005001           ,TZA     ,       EXCHANGE SWITCH
033010 054147  DFLT     ,STA     ,TSWIT  USING SENSE SWITCH 3.
033011 002000           ,CALL    ,TTYEX  SET UP I/O FOR PROPER DEVICE
033012 033021
033013 007400           ,ROF     ,
033014 005007           ,ZERO    ,07     ZERO ALL REGS AND CLEAR
033015 002000           ,CALL    ,SAVE   REGISTERS
033016 036143
033017 002000           ,CALL    ,GCOM   GO TO DEBUGGER
033020 035416

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 3

                     *
                     *   TELETYPE EXCHANGE ROUTINE:  CHANGES I/O DEVICE TO
                     *   OTHER TTY THAN ONE SELECTED.
                     *
                     *
033021 000000  TTYEX    ,ENTR    ,
033022 014135           ,LDA     ,TSWIT  IF THE TTY
033023 005211           ,CPA     ,       TRANSFER
033024 054133           ,STA     ,TSWIT  SWITCH IS
033025 001010           ,JAZ     ,TTYB   A -1 THEN
033026 033054
       033027  TTYA     ,EQU     ,*      SET UP FOR TTYA
033027 006010           ,LDAI    ,02000  SET UP INTERRUPT AREA
033030 002000
033031 050062           ,STA     ,PMIA
033032 006010           ,LDAI    ,INA
033033 033326
033034 050063           ,STA     ,PMIA+1
033035 006010           ,LDAI    ,MSKA   SET APROPRIATE MASK
033036 000375
033037 054121           ,STA     ,AMSK
033040 100440           ,EXC     ,0440   DISABLE
033041 010050           ,LDA     ,050
033042 006110           ,ORAI    ,0377-MSKB       OR IN OTHER'S BIT
033043 000100
033044 050050           ,STA     ,050
033045 100240           ,EXC     ,0240
033046 006010           ,LDAI    ,ODVA   OUTPUT DEVICE
033047 000004
033050 006020           ,LDBI    ,IDVA   INPUT DEVICE
033051 000004
033052 001000           ,JMP     ,TTY2
033053 033077
       033054  TTYB     ,EQU     ,*
033054 006010           ,LDAI    ,02000  SET UP INT LOC
033055 002000
033056 050074           ,STA     ,PMIB
033057 006010           ,LDAI    ,INA
033060 033326
033061 050075           ,STA     ,PMIB+1
```

```
105     033062 006010          ,LDAI    ,MSKB
        033063 000277
106     033064 054074          ,STA     ,AMSK
107     033065 100440          ,EXC     ,0440
108     033066 010050          ,LDA     ,050
109     033067 006110          ,ORAI    ,0377-MSKA      MASK OTHERS INT
        033070 000002
110     033071 050050          ,STA     ,050
111     033072 100240          ,EXC     ,0240
112     033073 006010          ,LDAI    ,ODVB
        033074 000001
113     033075 006020          ,LDBI    ,IDVB
        033076 000001
114     033077 006120   TTY2   ,ADDI    ,0101100        CHANGE:  OUTPUT SENSE IN
        033100 101100
115     033101 054244          ,STA     ,SENO
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 4

116     033102 006120          ,ADDI    ,02000  OUTPUT CHAR INSTRUCTION
        033103 002000
117     033104 054272          ,STA     ,OAR1
118     033105 054170          ,STA     ,OAR2
119     033106 005021          ,TBA     ,       GET INPUT DEVICE NUMBER
120     033107 006120          ,ADDI    ,0102500        AND ALL INPUT CHAR INSTR
        033110 102500
121     033111 054060          ,STA     ,CIA1
122     033112 054215          ,STA     ,CIA2
123     033113 006057          ,STAE    ,CIA3
        033114 034570
124     033115 001000          ,JMP*    ,TTYEX  RETURN
        033116 133021
125
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 5

126                     *
127     033117 000000   ON     ,ENTR    ,       TURN ON OUR INTERRUPT
128     033120 100440          ,EXC     ,0440   DISABLE
129     033121 054040          ,STA     ,TMP
130     033122 010050          ,LDA     ,050    GET MASK
131     033123 154035          ,ANA     ,AMSK   OUR MASK
132     033124 050050          ,STA     ,050
133     033125 103140          ,OAR     ,040
134     033126 014033          ,LDA     ,TMP
135     033127 100240          ,EXC     ,0240   ENABLE PIM
136     033130 001000          ,JMP*    ,ON     RETURN
        033131 133117
137                     *
138     033132 000000   OFF    ,ENTR    ,       TURN OFF OUR INTERRUPTS
139                     *                       OTHERS REMAIN ON
140     033133 100440          ,EXC     ,0440   DISABLE
141     033134 054025          ,STA     ,TMP
142     033135 014023          ,LDA     ,AMSK
143     033136 005211          ,CPA     ,       COMPL OF OUR MASK
144     033137 110050          ,ORA     ,050
145     033140 050050          ,STA     ,050
146     033141 103140          ,OAR     ,040
147     033142 014017          ,LDA     ,TMP
148     033143 100240          ,EXC     ,0240   ENABLE PIM
149     033144 001000          ,JMP*    ,OFF
        033145 133132
150                     *
151     033146 000000   WEC    ,ENTR    ,       CHANGE INT VECTOR ADDR TO B-REG
152     033147 014010          ,LDA     ,TSWIT
```

```
033150 001010          ,JAZ    ,WEC2
033151 033155
033152 060063          ,STB    ,PMIA+1 DEVICE A
033153 001000          ,JMP*   ,WEC
033154 133146
033155 060075  WEC2    ,STB    ,PMIB+1 DEVICE B
033156 001000          ,JMP*   ,WEC    DONE
033157 133146
               *
               *
033160 000000  TSWIT   ,DATA   ,0      TTY TRANSFER SWITCH
033161 000375  AMSK    ,DATA   ,MSKA   ACTUAL PIM MASK
033162 000000  TMP     ,DATA   ,0
               *
               *   COMMAND T
               *     ALLOWS USER TO CHANGE I/O DEVICE DURING EXECUTION
               *
033163 000000  TTY     ,ENTR   ,
033164 002000          ,CALL   ,TTYEX  CALL TELETYPE EXCHANGER
033165 033021
033166 002000          ,CALL   ,GCOM   RETURN
033167 035416

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 6

               *
               *
               *INPUT CONTROLLER AND I/O SUBROUTINES
               *TTY INTERRUPTS COME HERE WHEN RUNNING USER PROGRAM
               *
               *
033170 000000  INP     ,ENTR   ,       I/O CONTROLLER
033171 054152          ,STA    ,A      SAVE PRES A VALUE
033172 102504  CIA1    ,CIA    ,IDVA   INPUT FROM DEVICE IDVA INITIALLY
033173 002000          ,CALL   ,OFF
033174 033132
033175 006110          ,ORAI   ,0200   PUT IN PARITY BIT
033176 000200
033177 002000          ,CALL   ,LCASE  CHECK FOR LOWER CASE
033200 033247
033201 006147          ,SUBE   ,$I     IF CONTROL-N THEN
033202 036535
033203 001010          ,JAZ    ,DBUG   PREPARE TO ENTER DEBUGGER
033204 033212
       033205  DBG2    ,EQU    ,*      RESTORE AND RETURN
033205 014136          ,LDA    ,A
033206 002000          ,CALL   ,ON     OUR INTS ON
033207 033117
033210 001000          ,JMP*   ,INP            RETURN TO USER'S PROGRAM
033211 133170
       033212  DBUG    ,EQU    ,*
033212 006017          ,LDAE   ,INP    GET ADDR WHERE INTERRUPTED
033213 033170
033214 006147          ,SUBE   ,$H     WAS IT IN OUR AREA?
033215 036534
033216 001002          ,JAP    ,DBG2   NO, JMP TO RETURN
033217 033205
               *       INTERRUPT WAS FROM OUR AREA, GET SET FOR DEBUGGE
033220 014123          ,LDA    ,A      SET VARIABLES IN TABLE TO
033221 002000          ,CALL   ,SAVE           USER'S VALUES.
033222 036143
033223 006020          ,LDBI   ,INA
033224 033326
```

```
      033225 002000          ,CALL    ,WEC     RESET INT VECTOR
      033226 033146
      033227 006017          ,LDAE    ,INP
      033230 033170
      033231 006057          ,STAE    ,$P
      033232 036544
      033233 002000          ,CALL    ,CRLF    OUTPUT <*> AND GO TO DEBUGGER
      033234 033545
      033235 006010          ,LDAI    ,'<*'
      033236 136252
      033237 002000          ,CALL    ,COUT
      033240 037204
      033241 006010          ,LDAI    ,'>'
      033242 000276
      033243 002000          ,CALL    ,OUT
      033244 033345
      033245 002000          ,CALL    ,GCOM    TO DEBUGGER, NO RETURN
      033246 035416
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 7

                       *
                       *      CHECK A-REG FOR LOWER CASE LETTER AND CHANGE TO
                       *      UPPER CASE IF NECESSARY. JUNE 30  1971. GLH
                       *
      033247 000000 LCASE    ,ENTR    ,
      033250 006140          ,SUBI    ,0341
      033251 000341
      033252 001004          ,JAN     ,*+6
      033253 033260
      033254 006120          ,ADDI    ,0301
      033255 000301
      033256 001000          ,JMP     ,*+4
      033257 033262
      033260 006120          ,ADDI    ,0341
      033261 000341
      033262 001000          ,JMP*    ,LCASE
      033263 133247

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 8

                       *
                       *        IN CONTAINS THE IDLE LOOP FOR DEBUGGER INPUT
                       *        WHEN USER IS TALKING DIRECTLY TO ISADORA.
                       *        ROUTINE 'INA' GETS THE INTERRUPT AND STORES THE
                       *        CHARACTER IN 'CHAR'.
                       *
      033264 000000 IN       ,ENTR    ,        INPUT CHAR FOR DEBUGGER
      033265 005001          ,TZA     ,
      033266 054054          ,STA     ,CHAR
      033267 002000          ,CALL    ,ON      ENABLE OUR INT
      033270 033117
      033271 005000          ,NOP     ,
      033272 005000          ,NOP     ,        IDLE LOOP FOR INTRPT
      033273 014047          ,LDA     ,CHAR
      033274 001010          ,JAZ     ,*-3     IDLE
      033275 033271
      033276 103104 OAR2     ,OAR     ,ODVA    ECHO
      033277 006140          ,SUBI    ,0212
      033300 000212
      033301 001010          ,JAZ     ,IN2     LF
      033302 033311
      033303 006140          ,SUBI    ,3
      033304 000003
```

```
033305 001010            ,JAZ     ,IN4     CR
033306 033317
033307 001000            ,JMP     ,IN6
033310 033323
033311 006010  IN2       ,LDAI    ,0106615        ECHO CR CR WITH LF
033312 106615
033313 002000            ,CALL    ,COUT
033314 037204
033315 001000            ,JMP     ,IN6
033316 033323
033317 006010  IN4       ,LDAI    ,0212    ECHO LF WITH CR
033320 000212
033321 002000            ,CALL    ,OUT
033322 033345
033323 014017  IN6       ,LDA     ,CHAR    ORIGINAL CHAR
033324 001000            ,JMP*    ,IN      RETURN
033325 133264

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 9

               *
               *         DEBUGGER INTERRUPTS NORMALLY COME HERE
               *         WHILE ISADORA IS RUNNING
               *
033326 000000  INA       ,ENTR    ,
033327 054014            ,STA     ,A       SAVE AREG
033330 102504  CIA2      ,CIA     ,IDVA    INPUT FROM DEVICE IDVA INITIALLY
033331 002000            ,CALL    ,OFF     OUR INTS OFF, OTHERS ON
033332 033132
033333 006110            ,ORAI    ,0200    OR IN PARITY BIT
033334 000200
033335 002000            ,CALL    ,LCASE   CHECK FOR LOWER CASE LETTER
033336 033247
033337 054003            ,STA     ,CHAR
033340 014003            ,LDA     ,A       RESTORE AREG
033341 001000            ,JMP*    ,INA     RETURN
033342 133326
               *
033343 000000  CHAR      ,DATA    ,0
033344 000000  A         ,DATA    ,0       SAVE AREA FOR A REG
               *
033345 000000  OUT       ,ENTR    ,        OUTPUT CHAR FOR DEBUGGER
033346 101104  SENO      ,SEN     ,0100+ODVA,*+5  OUTPUT TO DEVICE ODVA IN
033347 033353
033350 005000            ,NOP     ,
033351 001000            ,JMP     ,*-3
033352 033346
033353 054027            ,STA     ,OUTA
033354 006150            ,ANAI    ,0377
033355 000377
033356 006140            ,SUBI    ,0212    LF?
033357 000212
033360 001010            ,JAZ     ,OUT2
033361 033376
033362 006140            ,SUBI    ,0215-0212      CR
033363 000003
033364 001010            ,JAZ     ,OUT2
033365 033376
033366 006140            ,SUBI    ,' '-0215
033367 000023
033370 001002            ,JAP     ,OUT2    >=BLANK
033371 033376
033372 006010            ,LDAI    ,'?'     MAKE A ?
```

```
       033373 000277
276    033374 001000            ,JMP     ,OAR1
       033375 033377
277    033376 014004  OUT2      ,LDA     ,OUTA    RESTORE CHAR
278    033377 103104  OAR1      ,OAR     ,ODVA
279    033400 014002            ,LDA     ,OUTA    RESTORE AREG
280    033401 001000            ,JMP*    ,OUT
       033402 133345
281    033403 000000  OUTA      ,DATA    ,0       SAVE AREG
282
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 10

283                   *
284                   *
285                   *PARAMETER TABLE
286                   *
287           033404  CMPT      ,EQU     ,*
288    033404 035667  $$DISP    ,DATA    ,GLCA,GTYP,DSLR < OR /
       033405 035721
       033406 034243
289    033407 036072  $$EQ      ,DATA    ,GVAR,GEXP,STVR =
       033410 035640
       033411 034403
290    033412 035667  $$SET     ,DATA    ,GLCA,DSLR      >
       033413 034243
291    033414 035640  $$QU      ,DATA    ,GEXP,GTYP,DEXP ?
       033415 035721
       033416 034414
292    033417 035651  $$AN      ,DATA    ,GLCE,GLCF,GEXP,GTYP,ALNT A,N
       033420 035575
       033421 035640
       033422 035721
       033423 034451
293    033424 035622  $$B       ,DATA    ,GNUM,GLCA,GEXP,BKPT B
       033425 035667
       033426 035640
       033427 034665
294    033430 035622  $$C       ,DATA    ,GNUM,CLR       C
       033431 035335
295    033432 035651  $$F       ,DATA    ,GLCE,GLCF,GEXP,FILL F
       033433 035575
       033434 035640
       033435 034647
296    033436 035704  $$G       ,DATA    ,GLCP,GO        G
       033437 035230
297    033440 034156  $$R       ,DATA    ,PREG   R
298    033441 033554  $$S       ,DATA    ,STEP   S
299    033442 034141  $$CMNT    ,DATA    ,CMNT   :
300    033443 035640  $$L       ,DATA    ,GEXP,BLDE      L
       033444 034133
301    033445 035651  $$D       ,DATA    ,GLCE,GLCF,GEXP,BDMP D
       033446 035575
       033447 035640
       033450 034423
302    033451 033163  $$T       ,DATA    ,TTY    T
303    033452 035651  $$M       ,DATA    ,GLCE,GLCF,GLCE,MOVE M
       033453 035575
       033454 035651
       033455 034617
304
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 11

305                   *
```

```
                  *COMMAND TABLE - HOLDS POINTERS TO CMPT
                  * TO GET AN INDEX INTO THIS TABLE  TAKE THE COMMAND CHAR
                  * AND SUBTRACT ':'. THE ENTRY AT THAT LOCATION IN THIS
                  * TABLE POINTS TO THE PARAMETER TABLE ABOVE.
                  *
033456 133442    CMT      ,DATA    ,($$CMNT)*      :
033457 000000             ,DATA    ,0              ;
033460 133404             ,DATA    ,($$DISP)*      <
033461 133407             ,DATA    ,($$EQ)*        =
033462 133412             ,DATA    ,($$SET)*       >
033463 133414             ,DATA    ,($$QU)*        ?
033464 000000             ,DATA    ,0              @
033465 133417             ,DATA    ,($$AN)*        A
033466 133424             ,DATA    ,($$B)*         B
033467 133430             ,DATA    ,($$C)*         C
033470 133445             ,DATA    ,($$D)*         D
033471 000000             ,DATA    ,0              E
033472 133432             ,DATA    ,($$F)*         F
033473 133436             ,DATA    ,($$G)*         G
033474 000000             ,DATA    ,0              H
033475 000000             ,DATA    ,0              I
033476 000000             ,DATA    ,0              J
033477 000000             ,DATA    ,0              K
033500 133443             ,DATA    ,($$L)*         L
033501 133452             ,DATA    ,($$M)*         M
033502 133417             ,DATA    ,($$AN)*        N
033503 000000             ,DATA    ,0              O
033504 000000             ,DATA    ,0              P
033505 000000             ,DATA    ,0              Q
033506 133440             ,DATA    ,($$R)*         R
033507 133441             ,DATA    ,($$S)*         S
033510 133451             ,DATA    ,($$T)*         T
033511 000000             ,DATA    ,0              U

```

```
                  *
                  *
                  *ERRORS IN EXECUTION SUBROUTINES
                  *
                  *
                  *
       033512    DERR     ,EQU     ,*
033512 006010    LERR     ,LDAI    ,' ?'    ADDRESS < LO
033513 120277
033514 002000             ,CALL    ,COUT
033515 037204
033516 006010             ,LDAI    ,'<L'
033517 136314
033520 002000             ,CALL    ,COUT
033521 037204
033522 001000             ,JMP     ,BANG
033523 035567
                  *
033524 006010    HERR     ,LDAI    ,' ?'    ADDRESS > HI
033525 120277
033526 002000             ,CALL    ,COUT
033527 037204
033530 006010             ,LDAI    ,'>H'
033531 137310
033532 002000             ,CALL    ,COUT
033533 037204
033534 001000             ,JMP     ,BANG
```

```
033535 035567
                *
                *
                *
                *SPECIAL OUTPUT SUBROUTINES
                *
033536 000000   CBK      ,ENTR     ,         OUTPUT A :
033537 006010            ,LDAI     ,':'
033540 000272
033541 002000            ,CALL     ,OUT
033542 033345
033543 001000            ,JMP*     ,CBK      RETURN
033544 133536
                *
033545 000000   CRLF     ,ENTR     ,         OUTPUT A CR/LF
033546 006010            ,LDAI     ,0106612
033547 106612
033550 002000            ,CALL     ,COUT
033551 037204
033552 001000            ,JMP*     ,CRLF     RETURN
033553 133545
```

```
                *
                *
                *STEP - INSTRUCTION EXECUTION SUBROUTINE
                *STEP EXECUTES THE INSTRUCTION WHOSE 1ST WORD IS STORED
                *THE LOCATION GIVEN BY P.  UPON EXECUTION THE B AND A RE
                *CONTAIN THE 1ST AND 2ND WORDS OF THE INSTRUCTION(S) RES
                *TIVELY.
033554 000000   STEP     ,ENTR     ,
033555 006037            ,LDXE     ,$P       LOAD X REG WITH P
033556 036544
033557 025000            ,LDB      ,0,1      LOAD B REG WITH WORD AT P
033560 004151            ,LSRB     ,9        CHECK FOR HALT INSTR
033561 001020            ,JBZ      ,HALT     JMP IF HALT
033562 034063
033563 025000            ,LDB      ,0,1      RELOAD
033564 015001            ,LDA      ,1,1      LOAD A REG WITH WORD AT P+1
033565 002000            ,CALL     ,STPP     STEP
033566 033571
033567 002000            ,CALL     ,GCOM     RETURN
033570 035416
                *
                *STPP - STEP EXECUTION ROUTINE
                *STPP EXECUTES THE INSTRUCTION WHOSE 1ST WORD IS IN THE
                *B REG AND WHOSE 2ND WORD IF ANY  IS IN THE A REG AS IF
                *WERE LOCATED AT P.  SUBROUTINE IS USED BY STEP AND BREA
                *
033571 000000   STPP     ,ENTR     ,
033572 054243            ,STA      ,INST+1 SAVE INSTRUCTION DATA
033573 054234            ,STA      ,SCND
033574 064240            ,STB      ,INST
                *        CHECK FOR SEN, IME, OME
033575 005021            ,TBA      ,         GET 1ST WORD OF INST
033576 004346            ,LSRA     ,6        LEFT 10 BITS
033577 005014            ,TAX      ,         SAVE
033600 006140            ,SUBI     ,01020    IS IT 1020XX (IME) ?
033601 001020
033602 001010            ,JAZ      ,XME      YES
033603 033633
033604 006140            ,SUBI     ,010      IS IT 1030XX (OME) ?
```

```
033605 000010
033606 001010            ,JAZ    ,XME    YES
033607 033633
033610 005041            ,TXA    ,       GET BACK INST (LEFT 10 BITS)
033611 004343            ,LSRA   ,3      LEFT 7 BITS NOW
033612 006140            ,SUBI   ,0101   IS IT 101XXX (SEN) ?
033613 000101
033614 001010            ,JAZ    ,JMP    YES, TREAT LIKE JMP INST
033615 033703
                   *     OTHERWISE CHECK SOME MORE TO SEPARATE INST TYPES
033616 005001            ,TZA    ,
033617 004444            ,LLRL   ,4
033620 001010            ,JAZ    ,OPZ    DOUBLE WORD INSTRUCTION
033621 033655
                   *     TEST IF ADDRESS RELATIVE TO P
033622 005001            ,TZA    ,
033623 004443            ,LLRL   ,3
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 14

033624 144304            ,SUB    ,VIER
033625 001010            ,JAZ    ,REL    IF A=0 THEN REL TO P
033626 033637
                   *     SET NO-OP AS 2ND WORD
033627 006010   BACK     ,LDAI   ,05000
033630 005000
033631 001000            ,JMP    ,BAC1   PROCESS INSTRUCTION
033632 034031
                   *
                   *     INST WAS IME OR OME, BUMP P-REG AND GO
033633 006047   XME      ,INRE   ,$P
033634 036544
033635 001000            ,JMP    ,BAC2
033636 034033
                   *
                   *     RELATIVE ADDRESS  MAKE TWO WORD DIRECT
033637 014175   REL      ,LDA    ,INST
033640 006150            ,ANAI   ,0777
033641 000777
033642 006127            ,ADDE   ,$P
033643 036544
033644 054171            ,STA    ,INST+1
033645 044170            ,INR    ,INST+1
033646 014166            ,LDA    ,INST
033647 004351            ,LSRA   ,9
033650 006110            ,ORAI   ,06007
033651 006007
033652 054162            ,STA    ,INST
033653 001000            ,JMP    ,BAC2
033654 034033
                   *     OP-CODE IS ZERO;  TEST IF SINGLE NON-ADDRESS
033655 004443   OPZ      ,LLRL   ,3
033656 001010            ,JAZ    ,BACK
033657 033627
                   *     TEST IF JUMP TYPE (<3)
033660 144251            ,SUB    ,THRE
033661 001004            ,JAN    ,JMP
033662 033703
                   *     TEST IF EXECUTE (=3)
033663 001010            ,JAZ    ,EXEC
033664 033672
                   *     TEST IF EXTENDED (=6)
033665 144244            ,SUB    ,THRE
033666 001010            ,JAZ    ,EXT
```

```
033667 033772
                  *        MUST BE SINGLE
033670 001000              ,JMP     ,BACK
033671 033627
                  *        EXECUTE TYPE  INCREMENT P FOR DOUBLE WORD
033672 006047     EXEC     ,INRE    ,$P
033673 036544
033674 002000              ,CALL    ,ADDR   GET OPERAND ADDRESS
033675 034016
033676 054137              ,STA     ,INST+1 STORE EFFECTIVE ADDR IN 2ND WORD
033677 006047              ,INRE    ,STIF   NO CHECK FOR STORE, INR IMMED
033700 035334
```

```
033701 001000              ,JMP     ,BAC2   EXECUTE SETUP INSTRUCTION
033702 034033
                  *        JUMP TYPE  TEST IF JUMP AND MARK
033703 006047     JMP      ,INRE    ,$P
033704 036544
033705 005111              ,IAR     ,
033706 001010              ,JAZ     ,JMPM
033707 033714
                  *        SET RETURN FOR JUMP
033710 006010              ,LDAI    ,JMPR
033711 033720
033712 001000              ,JMP     ,BAC1
033713 034031
                  *        SET RETURN FOR JUMP AND MARK
033714 006010     JMPM     ,LDAI    ,JMPS
033715 033755
033716 001000              ,JMP     ,BAC1
033717 034031
                  *        RETURN FROM JUMP;  PROCESS POSSIBLE INDIRECT
033720 002000     JMPR     ,CALL    ,ADDR
033721 034016
033722 006057              ,STAE    ,$P
033723 036544
033724 034205              ,LDX     ,THRE   TEST TO MAKE SURE IT DOES
033725 006015              ,LDAE    ,BKRT,1 JUMP INTO BKPT ROUTINE
033726 035023
033727 006147              ,SUBE    ,$P
033730 036544
033731 001010              ,JAZ     ,JPBK
033732 033740
033733 001040              ,JXZ*    ,STPP
033734 133571
033735 005344              ,DXR     ,
033736 001000              ,JMP     ,JMPR+5
033737 033725
033740 006077     JPBK     ,STXE    ,CNT
033741 035303
033742 006017              ,LDAE    ,STPP
033743 033571
033744 054007              ,STA     ,*+8
033745 006015              ,LDAE    ,BLOC,1
033746 035310
033747 006057              ,STAE    ,$P
033750 036544
033751 002000              ,CALL    ,BKP
033752 035115
033753 001000              ,JMP*    ,0
033754 100000
                  *        RETURN FROM JUMP AND MARK;  PROCESS INDIRECT
```

```
033755 000000  JMPS     ,BSS     ,1
033756 002000           ,CALL    ,ADDR
033757 034016
               *        STORE MARK IN JUMP ADDRESS  SET P
033760 005014           ,TAX     ,
033761 006017           ,LDAE    ,$P
033762 036544
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 16

033763 005111           ,IAR     ,
033764 055000           ,STA     ,0,1
033765 005144           ,IXR     ,
033766 006077           ,STXE    ,$P
033767 036544
033770 001000           ,JMP*    ,STPP
033771 133571
               *        EXTENDED ADDRESSING; TEST IF RELATIVE
033772 006047  EXT      ,INRE    ,$P
033773 036544
033774 004446           ,LLRL    ,6
033775 005001           ,TZA     ,
033776 004443           ,LLRL    ,3
033777 144131           ,SUB     ,VIER
034000 001010           ,JAZ     ,EREL
034001 034004
034002 001000           ,JMP     ,BAC2
034003 034033
               *        IS RELATIVE  MAKE DIRECT
034004 014030  EREL     ,LDA     ,INST
034005 006110           ,ORAI    ,7
034006 000007
034007 054025           ,STA     ,INST
034010 014025           ,LDA     ,INST+1
034011 006127           ,ADDE    ,$P
034012 036544
034013 005111           ,IAR     ,
034014 001000           ,JMP     ,BAC1
034015 034031
               *GO THROUGH POSSIBLE INDIRECT ADDRESS CHAIN  STARTING IN
               *        LEAVE RESULT IN A REG.
034016 000000  ADDR     ,ENTR    ,
034017 014010           ,LDA     ,SCND
034020 001002  ADRR     ,JAP*    ,ADDR
034021 134016
034022 006150           ,ANAI    ,077777
034023 077777
034024 005014           ,TAX     ,
034025 015000           ,LDA     ,0,1
034026 001000           ,JMP     ,ADRR
034027 034020
               *        DATA FOR STPP
034030 000000  SCND     ,BSS     ,1
               *        STEP EXECUTION
               *        STORE REVISED SECOND WORD
034031 054004  BAC1     ,STA     ,INST+1
034032 005000           ,NOP     ,
               *        RESTORE REGISTERS
034033 002000  BAC2     ,CALL    ,LOAD   LOAD ALL REGS AND OVFLW
034034 036154
               *        STEP THROUGH INSTRUCTION
034035 000000  INST     ,BSS     ,2
               *        SAVE REGISTERS; INCREMENT P; RETURN
034037 002000           ,CALL    ,SAVE   SAVE ALL REGS AND OVFLW
```

```
034040 036143
034041 006047                    ,INRE    ,$P
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 17

034042 036544
                        *        TEST TO SEE IF STEP INST WAS A STORE OR INR IMMD
034043 006017                    ,LDAE    ,STIF   IF IN BKP IGNORE TEST
034044 035334
034045 001010                    ,JAZ     ,*+4
034046 034051
034047 001000                    ,JMP*    ,STPP
034050 133571
034051 002000                    ,CALL    ,TSTI   TEST IF STI OR INRI
034052 034075
034053 001020                    ,JBZ*    ,STPP   NO RETURN
034054 133571
034055 006037                    ,LDXE    ,$P     ELSE STORE 2ND WORD OF INST
034056 036544
034057 005344                    ,DXR     ,       BACK INTO USER'S PROGRAM
034060 055000                    ,STA     ,0,1
034061 001000                    ,JMP*    ,STPP   RETURN
034062 133571
                        *
              034063    HALT     ,EQU     ,*      TRIED TO STEP A HALT INSTR
034063 006010                    ,LDAI    ,'HA'
034064 144301
034065 002000                    ,CALL    ,COUT   PRINT MSG
034066 037204
034067 006010                    ,LDAI    ,'LT'
034070 146324
034071 002000                    ,CALL    ,COUT
034072 037204
034073 002000                    ,CALL    ,PREG   PRINT REGS, NO RETURN
034074 034156
                        *

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 18

                        *
                        *TSTI - TEST IF A STORE OR INCR IMMEDIATE INSTRUCTION
                        *
034075 000000           TSTI     ,ENTR    ,
034076 005002                    ,TZB     ,       CLEAR B REG AS FLAG
034077 006017                    ,LDAE    ,INST   LOAD A REG WITH INST
034100 034035
034101 006067                    ,STBE    ,INST   CLEAR INST IN CASE OF 2ND PASS
034102 034035
034103 006140                    ,SUBI    ,06000  IF INST<06000 THEN NOT
034104 006000
034105 001004                    ,JAN*    ,TSTI   AN EXT OR IMM INST
034106 134075
034107 006140                    ,SUBI    ,0100   IF INSTSTILL + THEN CANNOT BE
034110 000100
034111 001002                    ,JAP*    ,TSTI   A LOAD STORE INR INST
034112 134075
034113 006120                    ,ADDI    ,040    IF INST<0 THEN INST IS A STORE
034114 000040
034115 001004                    ,JAN*    ,TSTI   OR INR EXT OR IMM
034116 134075
034117 154011                    ,ANA     ,VIER   IF 2 BIT ON THEN EXTENDED ELSE
034120 001010                    ,JAZ     ,*+4    IMMEDIATE
034121 034124
034122 001000                    ,JMP*    ,TSTI   EXTENDED  RETURN
```

```
034123 134075
034124 006017          ,LDAE   ,INST+1 IF IMM THEN LOAD A REG WITH
034125 034036
034126 005122          ,IBR    ,       INST OPER AND TURN ON FLAG
034127 001000          ,JMP*   ,TSTI   RETURN
034130 134075
                *
034131 000004  VIER    ,DATA   ,4      CONSTANT 4
034132 000003  THRE    ,DATA   ,3      CONSTANT 3

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 19

                *
                *
                *EXECUTION SUBROUTINES
                *
                *BLDE - LOAD A TAPE ON HIGH SPEED READER
                *COMMAND CHARACTER - L
                *SUBROUTINE LOADS A TAPE FROM THE HIGH SPEED READER
                *INTO CORE USING THE BINARY LOADER.
                *
034133 000000  BLDE    ,ENTR   ,
034134 005001          ,TZA    ,       CLEAR A REG FOR RETURN
034135 002000          ,CALL   ,BLD1   LOAD THE TAPE
034136 077630
034137 002000          ,CALL   ,GCOM   RETURN TO GET NEW COMMAND
034140 035416
                *
                *CMNT - COMMENT EXECUTION ROUTINE
                *COMMAND CHARACTER - :
                *ROUTINE ALLOWS ANYTHING TO BE TYPED AFTER COMMAND CHARA
                *TO TERMINATE COMMENT PRESS CR.  TO CONTINUE A COMMENT O
                *NEXT LINE PRESS LF.
                *
034141 000000  CMNT    ,ENTR   ,
034142 002000          ,CALL   ,IN     INPUT COMMENT CHAR
034143 033264
034144 006140          ,SUBI   ,0212   IS IT A LF
034145 000212
034146 002010          ,JAZM   ,GCOM   YES THEN END COMMENT
034147 035416
034150 006140          ,SUBI   ,03     IS IT A CR
034151 000003
034152 002010          ,JAZM   ,GCOM   YES THEN END COMMENT
034153 035416
034154 001000          ,JMP    ,CMNT+1 AND GO BACK AND GET NEXT CHAR
034155 034142
                *
                *PREG - PRINT ALL REGISTERS SUBROUTINE
                *COMMAND CHARACTER - R
                *SUBROUTINE PRINTS THE A B X AND P REGISTERS AND THE OVF
                *
034156 000000  PREG    ,ENTR   ,
034157 002000          ,CALL   ,CRLF   CR/LF BEFORE PRINTING
034160 033545
034161 006030          ,LDXI   ,3      USE X REG AS POINTER
034162 000003
034163 074055          ,STX    ,RIDX   TO SHOW REGISTER TO PRINT
034164 006015          ,LDAE   ,RTAB,1 GET REG TO OUTPUT
034165 034235
034166 002000          ,CALL   ,OUT    AND PRINT INDICATOR CHAR
034167 033345
034170 054051          ,STA    ,RTYP   SAVE IND CHAR TO GET INDEX
```

```
034171 002000           ,CALL    ,CBK     PRINT A : AND 2 BLKS
034172 033536
034173 014046           ,LDA     ,RTYP    GET IND CHAR
034174 006140           ,SUBI    ,'@'       GET INDEX IN VAR TABLE
034175 000300
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 20

034176 005014           ,TAX     ,        SET X REG = INDEX
034177 006015           ,LDAE    ,$,1     SET A REG TO VAR DATA
034200 036524
034201 002000           ,CALL    ,OTC     OUTPUT DATA IN OCTAL
034202 037117
034203 014176           ,LDA     ,TBK     OUTPUT 2 BLANKS FOR
034204 002000           ,CALL    ,COUT    SPACING
034205 037204
034206 034032           ,LDX     ,RIDX    LOAD X REG WITH IND PTR
034207 001040           ,JXZ     ,*+5     IF X=0 THEN PRINT OVFLW
034210 034214
034211 005344           ,DXR     ,        ELSE DEC PTR AND
034212 001000           ,JMP     ,PREG+5 GO BACK AND PROCESS NEXT REG
034213 034163
034214 006010           ,LDAI    ,'O'       LOAD AND PRINT OVFLW
034215 000317
034216 002000           ,CALL    ,OUT     IND CHAR
034217 033345
034220 002000           ,CALL    ,CBK     PRINT : AND 2 BKS
034221 033536
034222 006017           ,LDAE    ,$O      LOAD A REG WITH OVFLW
034223 036543
034224 001010           ,JAZ     ,*+3     IF O=0 THEN OVFLW OFF
034225 034227
034226 005101           ,INCR    ,1       ELSE SET O=1 TO INDICATE
034227 006120           ,ADDI    ,'0'
034230 000260
034231 002000           ,CALL    ,OUT
034232 033345
034233 002000           ,CALL    ,GCOM    RETURN
034234 035416
034235 000320  RTAB     ,DATA    ,'P','X','B','A'
034236 000330
034237 000302
034240 000301
034241 000003  RIDX     ,DATA    ,3
034242 000000  RTYP     ,DATA    ,0

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 21

               *
               *DSLR - DISPLAY AND SET LOCATION SUBROUTINE
               *COMMAND CHARACTERS DISPLAY - <  SET - >
               *SUBROUTINE PRINTS LOCATION VALUE AT LOCATION IF DISPLAY
               *WAITS FOR A VLUE TO BE SET TO THE LOCATION.  TO DEFAULT
               *BLANK FOLLOWED BY A CR OR LF IS NECESSARY.  INPUTTING J
               *A CR OR LF DOES NOTHING TO THE SPECIFIED LOCATION.
               *
034243 000000  DSLR     ,ENTR    ,
034244 006017           ,LDAE    ,COM     LOAD A REG WITH < OR >
034245 035533
034246 006140           ,SUBI    ,'>'     SUBTRACT OFF A >'
034247 000276
034250 005002           ,TZB     ,        CLEAR B REG TO IND FLAG
034251 001010           ,JAZ     ,*+3     IF A=0 THEN SET ELSE
034252 034254
```

```
034253 005122          ,IBR     ,        DISPLAY AND SET FLAG TO 1
034254 064123          ,STB     ,DSFG    SAVE FLAG FOR LATER
034255 006017          ,LDAE    ,P1      GET LOC PARM
034256 035527
034257 006057          ,STAE    ,$       UPDATE @
034260 036524
034261 054115          ,STA     ,DSLC    STORE AS INIT LOC FOR DISPLAY
034262 006017          ,LDAE*   ,$
034263 136524
034264 006057          ,STAE    ,$C
034265 036527
       034266  DISP    ,EQU     ,*
034266 006017          ,LDAE    ,$H      LOAD A REG WITH H
034267 036534
034270 144106          ,SUB     ,DSLC    SUBTRACT OFF LOC PARM
034271 001004          ,JAN     ,HERR    IF A<0 THEN H<S ERROR
034272 033524
034273 014103          ,LDA     ,DSLC    LOAD A REG WITH LOC TO PRINT
034274 002000          ,CALL    ,OTC     OUTPUT LOC IN OCTAL
034275 037117
034276 002000          ,CALL    ,CBK     OUTPUT A : FOLLOWED 2 BKS
034277 033536
034300 024077          ,LDB     ,DSFG    LOAD B REG WITH COM FLAG
034301 001020          ,JBZ     ,SET     IF B=0 THEN SET AND SKIP DISPLAY
034302 034312
034303 006017          ,LDAE*   ,DSLC    LOAD A REG WITH DATA AT LOC
034304 134377
034305 002000          ,CALL*   ,P2      AND OUTPUT IT IN SPECIFIED FORM
034306 135530
034307 014072          ,LDA     ,TBK     SPACE 2 BLANKS
034310 002000          ,CALL    ,COUT
034311 037204
034312 006017  SET     ,LDAE*   ,DSLC    LOAD A REG WITH DATA AT LOC
034313 134377
034314 054061          ,STA     ,SLOC    STORE AT SLOC AS DEFAULT ON SET
034315 002000          ,CALL    ,EXPR    CALL EXPR ANYZR TO PROCESS SET
034316 036255
034317 054061          ,STA     ,DSTP    SAVE RESULT OF ANALYSIS
034320 006017          ,LDAE    ,DEF     LOAD A REG WITH DEFAULT FLAG TO
034321 037025
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 22

034322 001004          ,JAN     ,*+4     SEE IF ANYTHING WAS PROCESSED
034323 034326
034324 014054          ,LDA     ,DSTP    IF NOTHING DONT SET LOC TO ANYTH
034325 054050          ,STA     ,SLOC    ELSE SET LOC TO NEW DATA
034326 005021          ,TBA     ,        LOAD A REG WITH SADD INDEX
034327 006140          ,SUBI    ,9       DEC BY 9 TO SEE IF BK LF OR CR
034330 000011
034331 001004          ,JAN     ,ROE     IF A< OR = 0 THEN CR OR LF OR ^
034332 034337
034333 005001          ,TZA     ,        ELSE A BLANK  SO ZERO A REG
034334 054041          ,STA     ,SLOC    AND STORE AS DFAULT IN SLOC
034335 001000          ,JMP     ,SET+3   GO BACK AND PROCESS NEW VALUE
034336 034315
034337 034036  ROE     ,LDX     ,SLOC    LOAD X REG WITH NEW LOC VALUE
034340 006077          ,STXE*   ,DSLC    STORE IT AS NEW LOC VALUE
034341 134377
034342 005021          ,TBA     ,
034343 006140          ,SUBI    ,7       SUB INDEX OF CRLF
034344 000007
034345 001004          ,JAN     ,DSFH    IF A<0 THEN CR DONE
034346 034365
```

```
034347 001010          ,JAZ     ,ROE2    LF
034350 034362
034351 014025          ,LDA     ,DSLC    MUST BE ^
034352 005311          ,DAR     ,        GO BACKWARDS
034353 054023          ,STA     ,DSLC
034354 006147          ,SUBE    ,$L      CHECK LOWER BOUND
034355 036540
034356 001004          ,JAN     ,LERR    TOO LOW
034357 033512
034360 001000          ,JMP     ,DISP    DISPLAY MORE
034361 034266
       034362 ROE2     ,EQU,*
034362 044014          ,INR     ,DSLC    INCR TO NEW LOC
034363 001000          ,JMP     ,DISP    JUMP TO PROCESS NEW LOC
034364 034266
034365 014011 DSFH     ,LDA     ,DSLC    LOAD A REG WITH FINAL LOC
034366 006057          ,STAE    ,$       UPDATE @ WITH NEW LOC
034367 036524
034370 006017          ,LDAE*   ,$
034371 136524
034372 006057          ,STAE    ,$C
034373 036527
034374 001000          ,JMP     ,GC3     RETURN
034375 035423
034376 000000 SLOC     ,DATA    ,0       NEW LOC DATA BUFFER
034377 000000 DSLC     ,DATA    ,0       LOC HOLDER
034400 000000 DSFG     ,DATA    ,0       FLAG TO SHOW COM  0=SET 1=DISPLA
034401 000000 DSTP     ,DATA    ,0       TEMP DATA HOLDER
034402 120240 TBK      ,DATA    ,'  '      2 BLANKS.  USED FOR SPACING.

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 23

              *
              *STVR - SET A VARIABLE
              *COMMAND CHARACTER - =
              *SUBROUTINE SETS A VARIABLE DESIGNATED BY THE INDEX IN
              *PARM 1 EQUAL TO THE VALUE IN PARM 2
              *
034403 000000 STVR     ,ENTR    ,
034404 006037          ,LDXE    ,P1      LOAD X REG WITH VARIABLE INDEX
034405 035527
034406 006017          ,LDAE    ,P2      LOAD A REG WITH NEW VAR DATA
034407 035530
034410 006055          ,STAE    ,$,1     STORE NEW DATA IN VARIABLE
034411 036524
034412 001000          ,JMP     ,GC3     RETURN
034413 035423
              *
              *DEXP - DISPLAY A VALUE OF AN EXPRESSION
              *COMMAND CHARACTER - ?
              *COMMAND DISPLAYS THE VALUE OF AN EXPRESSION IN PARM2 IN
              *FORMAT SPECIFIED BY PARM 1.
              *
034414 000000 DEXP     ,ENTR    ,
034415 006017          ,LDAE    ,P1      LOAD A REG WITH DISPLAY VALUE
034416 035527
034417 002000          ,CALL*   ,P2      OUTPUT IN SPECIFIED TYPE
034420 135530
034421 002000          ,CALL    ,GCOM    RETURN
034422 035416

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 24
```

```
              *
              *BDMP - BINARY DUMP ROUTINE
              *COMMAND CHARACTER - D
              *ROUTINE PUNCHES AN OBJECT TAPE USING THE BINARY DUMP
              *ROUTINE.  CONTENTS OF A B X REGS ARE SET BY THE
              *PARAMETER VALUES.
              *
034423 000000 BDMP     ,ENTR   ,
034424 002000          ,CALL   ,GCOM    NO DUMP ANYMORE
034425 035416
034426 006010          ,LDAI   ,0102204         SUPPRESS PRINT TEMPORARI
034427 102204
034430 002000          ,CALL   ,COUT
034431 037204
034432 006017          ,LDAE   ,P1      LOAD A REG WITH START LOC
034433 035527
034434 006027          ,LDBE   ,P2      LOAD B REG WITH END LOC
034435 035530
034436 006037          ,LDXE   ,P3      LOAD X REG WITH EXEC LOC
034437 035531
034440 007401          ,SOF    ,        SET OVFLW FOR PUNCH
034441 002000          ,CALL   ,BLD2    BINARY DUMP
034442 077434
034443 006010          ,LDAI   ,0201    TURN PRINT BACK ON
034444 000201
034445 002000          ,CALL   ,OUT
034446 033345
034447 002000          ,CALL   ,GCOM    GET NEW COMMAND
034450 035416

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 25

              *
              *A,N <START ADDR> <FINISH ADDR> <VALUE-LOOKING-FOR> <OUT
              *ALNT - ALL AND NOT EXECUTION SUBROUTINES
              *COMMAND CHARACTERS ALL - A  NOT - N
              *DISPLAYS ALL ADDRESSES & VALUES OF DATA THAT WHEN PARM
              *ARE ANDED EQUAL TO THE DATA AT THE GIVEN ADDRESS.  ERRO
              *DATA AND M ARE ANDED SHOULD EQUAL PARM 3.  ALL PRINTS A
              *OCCURRENCES OF EQUALITY AND NOT PRINTS ALL OCCURRENCES
              *INEQUALITY.
              *
034451 000000 ALNT     ,ENTR   ,
034452 005001          ,TZA    ,
034453 054123          ,STA    ,ACNT    ZERO COUNT
034454 002000          ,CALL   ,CRLF    CR/LF BEFORE PRINTING
034455 033545
034456 002000          ,CALL   ,PCHK    CHECK PARMS P1,P2
034457 034603
034460 006020          ,LDBI   ,ANP3    SET UP INT LOC
034461 034564
034462 002000          ,CALL   ,WEC
034463 033146
034464 006017          ,LDAE   ,COM     LOAD A REG WITH COM CHAR
034465 035533
034466 006140          ,SUBI   ,'N'     TO DETERMINE WHTHER ALL OR NOT
034467 000316
034470 005002          ,TZB    ,        CLEAR B REG FOR FLAG
034471 001010          ,JAZ    ,*+3     IF A=0 THEN NOT ELSE
034472 034474
034473 005322          ,DBR    ,        ALL.  SET FLAG = -1
034474 064103          ,STB    ,ANFG    SAVE FLAG FOR LATER
034475 006017          ,LDAE   ,P1      LOAD A REG WITH S(START LOC)
```

```
034476 035527
034477 054101          ,STA      ,ANLC    SAVE STARTING LOC
034500 002000          ,CALL     ,ON      OUR INTS ON
034501 033117
034502 024075  ANPS    ,LDB      ,ANFG    LOAD B REG WITH COM FLAG
034503 006017          ,LDAE*    ,ANLC    LOAD A REG WITH DATA AT LOC
034504 134601
034505 006157          ,ANAE     ,$M      AND WITH MASK M
034506 036541
034507 006147          ,SUBE     ,P3      SUBTRACT OFF VALUE
034510 035531
034511 001010          ,JAZ      ,*+6     IF A=0 THEN TEST IF ALL
034512 034517
034513 001020          ,JBZ      ,*+6     IF NOT THEN PRINT LOC
034514 034521
034515 001000          ,JMP      ,ANP2    ELSE DONT PRINT LOC
034516 034546
034517 001020          ,JBZ      ,ANP2    FOUND BUT NOT THEN SKIP PRINT
034520 034546
034521 014057          ,LDA      ,ANLC    IF VALID FOR OUTPUT THEN
034522 002000          ,CALL     ,OTC     LOAD A REG WITH LOC  PRINT IN
034523 037117
034524 002000          ,CALL     ,CBK        OUTPUT ':
034525 033536
034526 006017          ,LDAE*    ,ANLC    GET VALUE OF LOCATION
```

```
034527 134601
034530 002000          ,CALL*    ,P4      OUTPUT VALUE IN FORMAT
034531 135532
034532 014044          ,LDA      ,ACNT    ACNT = NUMBER OF WDS ON LINE
034533 005111          ,IAR      ,
034534 006150          ,ANAI     ,3       MODULO 4
034535 000003
034536 054040          ,STA      ,ACNT
034537 006017          ,LDAE     ,TBK     2 BLANKS
034540 034402
034541 002000          ,CALL     ,COUT
034542 037204
034543 014033          ,LDA      ,ACNT
034544 002010          ,JAZM     ,CRLF    4 WORDS ON THIS LINE
034545 033545
       034546  ANP2    ,EQU      ,*
034546 014032          ,LDA      ,ANLC    CHECK FOR TOO HIGH
034547 006147          ,SUBE     ,P2
034550 035530
034551 001002          ,JAP      ,ANP4    DONE
034552 034556
034553 044025          ,INR      ,ANLC    NEXT LOC
034554 001000          ,JMP      ,ANPS
034555 034502
       034556  ANP4    ,EQU      ,*
034556 006020          ,LDBI     ,INA
034557 033326
034560 002000          ,CALL     ,WEC     RESTORE REG INTERRUPT
034561 033146
034562 002000          ,CALL     ,GCOM    ENTER DEBUGGER
034563 035416
               *
034564 000000  ANP3    ,ENTR     ,        COME HERE ON INT OR WHEN DONE WI
034565 054014          ,STA      ,AA      SAVE A
034566 002000          ,CALL     ,OFF     OUR INTS OFF
034567 033132
```

```
034570 102504 CIA3    ,CIA     ,IDVA    CLEAR INPUT REG
034571 006017         ,LDAE    ,P2      LAST ADDR FOR SEARCH
034572 035530
034573 054005         ,STA     ,ANLC    CAUSE ALNT TO STOP NEXT TIME
034574 014005         ,LDA     ,AA      RESTORE A
034575 001000         ,JMP*    ,ANP3    RETURN
034576 134564
                *
034577 000000 ACNT    ,DATA    ,0       COUNT # WORDS ON LINE
034600 000000 ANFG    ,DATA    ,0       COMMAND FLAG  0=NOT  -1=ALL
034601 000000 ANLC    ,DATA    ,0       SEARCH LOC HOLDER
034602 000000 AA      ,DATA    ,0       SAVE A REG

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 27

              *
              *PCHK -- UTILITY ROUTINE TO CHECK IF P1 <= P2. IF NOT, T
              *        WE EXCHANGE P1 AND P2.
              *
034603 000000 PCHK    ,ENTR    ,
034604 014723         ,LDA     ,P2
034605 144721         ,SUB     ,P1
034606 001002         ,JAP*    ,PCHK    OK
034607 134603
034610 014716         ,LDA     ,P1
034611 005012         ,TAB     ,
034612 014715         ,LDA     ,P2
034613 054713         ,STA     ,P1
034614 064713         ,STB     ,P2      P1 AND P2 EXCHANGED
034615 001000         ,JMP*    ,PCHK    RETURN
034616 134603

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 28

              *
              *MOVE - M <START.SOURCE> <END.SOURCE> <START.TARGET>
              *       MOVE WORDS [P1,P2] TO [P3,P3+P2-P1]
              *       MUST HAVE P1<=P2, P2<=H, L<=P1, P3+P2-P1 <= H
              *
034617 000000 MOVE    ,ENTR    ,
034620 002000         ,CALL    ,PCHK            CHECK PARMS P1,P2
034621 034603
034622 014705         ,LDA     ,P2
034623 124705         ,ADD     ,P3
034624 144702         ,SUB     ,P1
034625 006147         ,SUBE    ,$H
034626 036534
034627 005311         ,DAR     ,
034630 001002         ,JAP     ,HERR            P3+P2-P1 (FINAL TARGET)
034631 033524
034632 034674         ,LDX     ,P1              MOVE IT
034633 005041 MV1     ,TXA     ,
034634 144673         ,SUB     ,P2
034635 005311         ,DAR     ,
034636 002002         ,JAPM    ,GCOM            FINISHED
034637 035416
034640 015000         ,LDA     ,0,1             SOURCE
034641 006057         ,STAE*   ,P3              TO TARGET
034642 135531
034643 005144         ,IXR     ,
034644 044664         ,INR     ,P3
034645 001000         ,JMP     ,MV1             NEXT WORD
034646 034633
```

```

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 29

                     *
                     *
                     *FILL - FILL LOCATIONS EXECUTION SUBROUTINE
                     *       F <START.LOC> <END.LOC> <VALUE>
                     *SUBROUTINES FILLS LOCATIONS PARM 1 THROUGH PARM2 WITH T
                     *VALUE IN PARM 3.
                     *
034647 000000        FILL    ,ENTR    ,
                     *       WE USE THE MOVE ROUTINE
                     *       P1 := START (P1)
                     *       P2 := FINISH (P2) -1
                     *       P3 := START (P1) +1
                     *       [P1] := VALUE (P3)
034650 002000                ,CALL    ,PCHK   CHECK THE PARMS P1,P2
034651 034603
034652 014655                ,LDA     ,P2
034653 005311                ,DAR     ,
034654 054653                ,STA     ,P2
034655 014653                ,LDA     ,P3
034656 006057                ,STAE*   ,P1
034657 135527
034660 014646                ,LDA     ,P1
034661 005111                ,IAR     ,
034662 054646                ,STA     ,P3
034663 002000                ,CALL    ,MOVE   NO RETURN
034664 034617

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 30

                     *
                     *
                     *BKPT - SET BREAK POINT SUBROUTINE
                     *SETS UP BREAKPOINT INFO.
                     *BLOC HOLDS BREAKPOINT LOCATION.
                     *BWD1 AND BWD2 HOLD INSTRUCTIONS REPLACED BY JUMP TO
                     *  BREAKPOINT PROCESSING.
                     *BCNT HOLDS COUNT OF TIMES THROUGH BREAKPOINT.
                     *BLMT HOLDS LIMIT ON TIMES THROUGH BREAKPOINT.
                     *BTYP HOLDS FLAG ON TYPES OF INSTRUCTIONS REPLACED.
                     *  -1 = DOUBLE WORD INSTRUCTION
                     *   0 = TWO SINGLE WORD INSTRUCTIONS
                     *   1 = SINGLE FOLLOWED BY A DOUBLE WORD INSTRUCTION
                     *CNT IIS USED ON ENTRY TO BREAKPOINT PROCESSING TO
                     *  INDICATE WHICH BREAKPOINT WAS ACTIVATED.
                     *
034665 000000        BKPT    ,ENTR    ,
034666 014640                ,LDA     ,P1     CHECK PARM 1
034667 001004                ,JAN     ,BKQ    IF NEG, LIST BRKPTS
034670 034770
034671 034166                ,LDX     ,TRES   TEST TO MAKE SURE A
       034672        BKP2    ,EQU     ,*
034672 006015                ,LDAE    ,BLOC,1 BRKPT HAS NOT BEEN SET
034673 035310
034674 144633                ,SUB     ,P2     AT THIS LOC PREVIOUSLY
034675 001010                ,JAZ     ,BERR   YES THEN ERROR
034676 035406
034677 001040                ,JXZ     ,BKP5
034700 034704
034701 005344                ,DXR     ,
034702 001000                ,JMP     ,BKP2
```

```
034703 034672
       034704  BKP5    ,EQU    ,*
034704 034622          ,LDX    ,P1
034705 006015          ,LDAE   ,BLOC,1 TEST TO SEE IF PREVIOUS BKPT
034706 035310
034707 001002          ,JAP    ,BERR   AT P1
034710 035406
034711 014617          ,LDA    ,P3
034712 006055          ,STAE   ,BLMT,1
034713 035330
034714 005001          ,TZA    ,
034715 006055          ,STAE   ,BCNT,1
034716 035324
034717 054363          ,STA    ,CNT
034720 024607          ,LDB    ,P2
034721 006067          ,STBE   ,$      UPDATE @
034722 036524
034723 016000          ,LDA    ,0,2
034724 006057          ,STAE   ,$C
034725 036527
034726 006065          ,STBE   ,BLOC,1
034727 035310
034730 016000          ,LDA    ,0,2
034731 006055          ,STAE   ,BWD1,1
034732 035314
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 31

034733 006010          ,LDAI   ,01000  LOAD A REG WITH JMP INSTR
034734 001000
034735 056000          ,STA    ,0,2
034736 016001          ,LDA    ,1,2
034737 006055          ,STAE   ,BWD2,1
034740 035320
034741 006015          ,LDAE   ,BKRT,1
034742 035023
034743 056001          ,STA    ,1,2
034744 006025          ,LDBE   ,BWD1,1
034745 035314
               *GET TYPE FLAG FOR 1ST WORD
034746 002000          ,CALL   ,TYPA
034747 035027
034750 006055          ,STAE   ,BTYP,1
034751 035304
               *EXIT IF DOUBLE WORD
034752 001004          ,JAN    ,BKPF
034753 034766
034754 006025          ,LDBE   ,BWD2,1
034755 035320
               *GET TYPE FLAG FOR 2ND WORD
034756 002000          ,CALL   ,TYPA
034757 035027
               *EXIT IF ANOTHER SINGLE WORD
034760 001010          ,JAZ    ,BKPF
034761 034766
               *SET FLAG FOR SINGLE THEN DOUBLE
034762 005111          ,IAR    ,
034763 005111          ,IAR    ,
034764 006055          ,STAE   ,BTYP,1
034765 035304
               *TERMINATE COMMAND AND RETURN
034766 002000  BKPF    ,CALL   ,GCOM
034767 035416
               *
```

```
                            *       OUTPUT LOCATIONS WITH BRKPTS ON THEM
                            *
        034770 005004       BKQ     ,TZX    ,        START AT BRKPT 0
        034771 006015               ,LDAE   ,BLOC,1 GET LOCATION ADDRESS
        034772 035310
        034773 001004               ,JAN    ,BKQ4    NOT A BREAK PT
        034774 035012
        034775 005041               ,TXA    ,
        034776 006120               ,ADDI   ,' 0'
        034777 120260
        035000 002000               ,CALL   ,COUT    OUTPUT #:
        035001 037204
        035002 002000               ,CALL   ,CBK
        035003 033536
        035004 074015               ,STX    ,BKQX    SAVE X
        035005 006015               ,LDAE   ,BLOC,1 GET BRK PT LOC
        035006 035310
        035007 002000               ,CALL   ,OTC     OUTPUT IT IN OCTAL
        035010 037117
        035011 034010               ,LDX    ,BKQX    RESTORE X
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 32

        035012 005144       BKQ4    ,IXR    ,        NEXT
        035013 005041               ,TXA    ,
        035014 006140               ,SUBI   ,4       DONE?
        035015 000004
        035016 001004               ,JAN    ,BKQ+1   NO
        035017 034771
        035020 002000               ,CALL   ,GCOM    BACK TO TOP LEVEL
        035021 035416
        035022 000000       BKQX    ,DATA   ,0
                            *
                            *DATA FOR SETTING JUMP
        035023 035064       BKRT    ,DATA   ,(BK0),(BK1),(BK2),(BK3)
        035024 035063
        035025 035062
        035026 035061

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 33

                            *
        035027 000000       TYPA    ,ENTR   ,
                            *TYPE ANALYSIS SUBROUTINE; 1ST WORD IN B REG; ON EXIT
                            *A REG HOLDS -1 IF DOUBLE  0 IF SINGLE.
        035030 005001               ,TZA    ,
        035031 004444               ,LLRL   ,4
                            *IF NON-ZERO OP IS SINGLE WORD SO EXIT ZERO
        035032 001010               ,JAZ    ,TYPB
        035033 035037
        035034 005001       TYPF    ,TZA    ,
        035035 001000               ,JMP*   ,TYPA
        035036 135027
                            *OP ZERO  IF M = 0 4 5 7 STILL SINGLE WORD
        035037 004443       TYPB    ,LLRL   ,3
        035040 001010               ,JAZ*   ,TYPA
        035041 135027
                            *TEST IF LESS THAN 4
        035042 144015               ,SUB    ,TRES
        035043 001010               ,JAZ    ,TYPD
        035044 035047
        035045 001002               ,JAP    ,TYPC
        035046 035053
                            *SET -1 AND EXIT
```

```
035047 005001 TYPD     ,TZA     ,
035050 005311          ,DAR     ,
035051 001000          ,JMP*    ,TYPA
035052 135027
              *M IS 4 OR GREATER; IF 6  SET DOUBLE WORD FLAG.
035053 144004 TYPC     ,SUB     ,TRES
035054 001010          ,JAZ     ,TYPD
035055 035047
035056 001000          ,JMP     ,TYPF
035057 035034
              *
035060 000003 TRES     ,DATA    ,3      CONSTANT 3

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 34

              *
              *        EXECUTED BREAKPOINTS COME HERE
              *        WE EXPECT CNT TO BE ZERO
035061 044221 BK3      ,INR     ,CNT
035062 044220 BK2      ,INR     ,CNT
035063 044217 BK1      ,INR     ,CNT
              *SAVE REGISTERS
035064 002000 BK0      ,CALL    ,SAVE   SAVE THE REGISTERS
035065 036143
              *TEST IF SHOULD TAKE BREAKPOINT
035066 034214          ,LDX     ,CNT
035067 006015          ,LDAE    ,BLOC,1
035070 035310
035071 006057          ,STAE    ,$P
035072 036544
035073 006015          ,LDAE    ,BCNT,1
035074 035324
035075 006145          ,SUBE    ,BLMT,1
035076 035330
035077 001002          ,JAP     ,BKA
035100 035200
              *DONT BREAK; EXECUTE BREAKPOINT INSTRUCTIONS
035101 006045          ,INRE    ,BCNT,1
035102 035324
035103 006020          ,LDBI    ,INP
035104 033170
035105 002000          ,CALL    ,WEC    SET UP INT LOC
035106 033146
035107 002000          ,CALL    ,BKP
035110 035115
035111 002000          ,CALL    ,ON     OUR INTS ON
035112 033117
035113 001000          ,JMP*    ,$P     RETURN TO USER PGM
035114 136544
              *
              *PROCESS BREAKPOINT; BK # IN X REG AND CNT
              *
035115 000000 BKP      ,ENTR    ,
035116 005001          ,TZA     ,       CLEAR DOUBLE WORD FLAG
035117 054214          ,STA     ,STIF
035120 006015          ,LDAE    ,BTYP,1
035121 035304
035122 006025          ,LDBE    ,BWD1,1
035123 035314
035124 001004          ,JAN     ,BDBL
035125 035164
035126 001010          ,JAZ     ,BSNG
035127 035171
```

```
                    *SINGLE FOLLOWED BY DOUBLE; STEP THROUGH SINGLE
035130 002000              ,CALL    ,STPP
035131 033571
                    *SET UP AND STEP THROUGH DOUBLE
035132 034150              ,LDX     ,CNT
035133 006025              ,LDBE    ,BWD2,1
035134 035320
035135 006035              ,LDXE    ,BLOC,1
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 35

035136 035310
035137 015002              ,LDA     ,2,1
035140 002000       BKB    ,CALL    ,STPP
035141 033571
                    *TEST TO SEE IF DOUBLE WORD.  IF SO  THEN STI OR INRI?
035142 034140              ,LDX     ,CNT     TEST IF DOUBLE WORD
035143 006015              ,LDAE    ,BTYP,1
035144 035304
035145 001002              ,JAP     ,BKD     NO THEN SKIP CHECK
035146 035155
035147 002000              ,CALL    ,TSTI    TEST TO SEE IF STI OR INRI
035150 034075
035151 001020              ,JBZ     ,BKD     IF NOT THEN CONTINUE PROCESSING
035152 035155
035153 006055              ,STAE    ,BWD2,1 ELSE UPDATE 2ND OF BKPT
035154 035320
                    *RESTORE REGISTERS AND GO
035155 005001       BKD    ,TZA     ,        CLEAR CNT AND DOUBLE WORD FLAG
035156 054155              ,STA     ,STIF
035157 054123              ,STA     ,CNT
035160 002000              ,CALL    ,LOAD    LOAD ALL REGS AND OVFLW
035161 036154
035162 001000              ,JMP*    ,BKP
035163 135115
                    *DOUBLE WORD INSTRUCTION
035164 006015       BDBL   ,LDAE    ,BWD2,1
035165 035320
035166 044145              ,INR     ,STIF    TURN ON DOUBLE WORD FLAG
035167 001000              ,JMP     ,BKB
035170 035140
                    *SINGLE WORD INSTRUCTIONS
035171 002000       BSNG   ,CALL    ,STPP
035172 033571
035173 034107              ,LDX     ,CNT
035174 006025              ,LDBE    ,BWD2,1
035175 035320
035176 001000              ,JMP     ,BKB
035177 035140
                    *TAKE BREAK; RESET COUNT TO ZERO
035200 005001       BKA    ,TZA     ,
035201 006055              ,STAE    ,BCNT,1
035202 035324
035203 002000              ,CALL    ,CRLF
035204 033545
                    *       PRINT BREAK POINT NUMBER THAT WAS EXECUTED
                    *       E.G. <0> FOR BREAKPOINT ZERO.
035205 006010              ,LDAI    ,'<'
035206 000274
035207 002000              ,CALL    ,OUT
035210 033345
035211 014071              ,LDA     ,CNT
035212 006110              ,ORAI    ,'0'
035213 000260
```

```
035214 002000            ,CALL    ,OUT
035215 033345
035216 006010            ,LDAI    ,'>'
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 36

035217 000276
035220 002000            ,CALL    ,OUT
035221 033345
035222 006020            ,LDBI    ,INA
035223 033326
035224 002000            ,CALL    ,WEC     SET UP INT ADDR
035225 033146
035226 002000            ,CALL    ,GCOM
035227 035416

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 37

                  *
                  *GO ROUTINE; CHECK IF BREAKPOINT
035230 000000 GO         ,ENTR    ,
035231 002000            ,CALL    ,CRLF
035232 033545
035233 034273            ,LDX     ,P1
035234 006077            ,STXE    ,$P
035235 036544
035236 006030            ,LDXI    ,4
035237 000004
035240 005344 BGOB       ,DXR     ,
035241 006015            ,LDAE    ,BLOC,1
035242 035310
035243 144263            ,SUB     ,P1
035244 001010            ,JAZ     ,BGOA
035245 035270
035246 001040            ,JXZ     ,*+4
035247 035252
035250 001000            ,JMP     ,BGOB
035251 035240
035252 006010            ,LDAI    ,GEND
035253 035264
035254 006057            ,STAE    ,BKP
035255 035115
035256 006020            ,LDBI    ,INP     POINT INPUT TO INPUT CNTRLR
035257 033170
035260 002000            ,CALL    ,WEC     INT LOC PTS TO INP
035261 033146
035262 001000            ,JMP     ,BKD
035263 035155
035264 002000 GEND       ,CALL    ,ON
035265 033117
035266 001000            ,JMP*    ,$P
035267 136544
035270 074012 BGOA       ,STX     ,CNT
035271 006020            ,LDBI    ,INP
035272 033170
035273 002000            ,CALL    ,WEC
035274 033146
035275 002000            ,CALL    ,BKP
035276 035115
035277 002000            ,CALL    ,ON
035300 033117
035301 001000            ,JMP*    ,$P      TO USER AGAIN
035302 136544
                  *DATA STORAGE FOR BREAKPOINT
```

```
035303 000000  CNT      ,BSS      ,1
035304 000000  BTYP     ,DATA     ,0,0,0,0
035305 000000
035306 000000
035307 000000
035310 177777  BLOC     ,MZE      ,-1,-1,-1,-1
035311 177777
035312 177777
035313 177777
035314 000000  BWD1     ,DATA     ,0,0,0,0
ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 38

035315 000000
035316 000000
035317 000000
035320 000000  BWD2     ,DATA     ,0,0,0,0
035321 000000
035322 000000
035323 000000
035324 000000  BCNT     ,DATA     ,0,0,0,0
035325 000000
035326 000000
035327 000000
035330 000000  BLMT     ,DATA     ,0,0,0,0
035331 000000
035332 000000
035333 000000
               *
035334 000000  STIF     ,DATA     ,0       DOUBLE WORD FLAG

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 39

               *
               *CLEAR BREAKPOINT SUBROUTINE
               *
035335 000000  CLR      ,ENTR,
035336 014170           ,LDA      ,P1
035337 001004           ,JAN      ,CLR2    JMP IF WE CLEAR ALL
035340 035345
035341 002000           ,CALL     ,CLEAR   CLEAR JUST ONE
035342 035364
035343 002000           ,CALL     ,GCOM    NO RETURN
035344 035416
035345 005001  CLR2     ,TZA      ,
035346 054160           ,STA      ,P1
035347 002000           ,CALL     ,CLEAR   0
035350 035364
035351 044155           ,INR      ,P1
035352 002000           ,CALL     ,CLEAR   1
035353 035364
035354 044152           ,INR      ,P1
035355 002000           ,CALL     ,CLEAR   2
035356 035364
035357 044147           ,INR      ,P1
035360 002000           ,CALL     ,CLEAR   3
035361 035364
035362 002000           ,CALL     ,GCOM    RETURN TO TOP LEVEL
035363 035416
               *
035364 000000  CLEAR    ,ENTR     ,
035365 034141           ,LDX      ,P1
035366 006015           ,LDAE     ,BLOC,1
035367 035310
```

```
035370 001004          ,JAN*    ,CLEAR  RETURN, NO BKPT SET
035371 135364
035372 005012          ,TAB     ,
035373 006015          ,LDAE    ,BWD1,1
035374 035314
035375 056000          ,STA     ,0,2
035376 006015          ,LDAE    ,BWD2,1
035377 035320
035400 056001          ,STA     ,1,2
035401 005301          ,DECR    ,1      PUT -1 IN BLOC
035402 006055          ,STAE    ,BLOC,1
035403 035310
035404 001000          ,JMP*    ,CLEAR  RETURN
035405 135364
                *
035406 002000   BERR   ,CALL    ,CRLF
035407 033545
035410 006010          ,LDAI    ,'PB'       PREVIOUS BKPT ERROR
035411 150302
035412 002000          ,CALL    ,COUT
035413 037204
035414 001000          ,JMP     ,BANG
035415 035567

ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 40

                *
                *
                *GCOM - COMMAND CONTROLLER AND PROCESSOR ROUTINE
                *SUBROUTINE INPUTS AND CHECKS COMMAND CHARACTERS.  PROCE
                *OF THE COMMAND BY GETTING THE PROPER PARAMETERSOFF THE
                *COMMAND POINTER TABLES(CMT AND CMPT) AND BRANCHING TO T
                *PROPER COMMAND EXECUTION SUBROUTINE.
                *
035416 000000   GCOM   ,ENTR    ,
035417 002000          ,CALL    ,ON     OUT INTS ON
035420 033117
035421 002000          ,CALL    ,CRLF   OUTPUT CR/LF TO TERMINATE PREVIO
035422 033545
       035423   GC3    ,EQU     ,*
035423 005002          ,TZB     ,       CLEAR B REG AND RESET
035424 006067          ,STBE    ,PDEF   PERMANENT DEFAULT FLAG
035425 037026
035426 007400          ,ROF     ,       RESET OVFLW IND IF IT WAS ON
035427 006010          ,LDAI    ,'#'     PROMPT SIGN
035430 000243
035431 002000          ,CALL    ,OUT    OUTPUT CHARACTER
035432 033345
035433 002000          ,CALL    ,IN     INPUT COMMAND CHAR
035434 033264
035435 054075          ,STA     ,COM    SAVE COM CHAR FOR LATER
035436 006140          ,SUBI    ,0203   IF CONTROL-C
035437 000203
035440 001010          ,JAZ     ,GMON   GO TO MON
035441 035517
035442 014070          ,LDA     ,COM    RESTORE COMMAND CHARACTER
035443 006140          ,SUBI    ,'/'    / IS SYNONYM FOR <
035444 000257
035445 001010          ,JAZ     ,GC4
035446 035452
035447 014063          ,LDA     ,COM
035450 001000          ,JMP     ,*+6
035451 035456
```

```
                035452  GC4      ,EQU     ,*
        035452  006010           ,LDAI    ,'<'
        035453  000274
        035454  001000           ,JMP     ,GC5
        035455  035464
        035456  002000           ,CALL    ,ILC,':','U'  TEST FOR LEGAL COM CHAR
        035457  036452
        035460  000272
        035461  000325
        035462  001001           ,JOF     ,BANG   IF NOT THEN ERROR
        035463  035567
                035464  GC5      ,EQU     ,*
        035464  006140           ,SUBI    ,':'     ELSE GET AN INDEX ON CMT
        035465  000272
        035466  005012           ,TAB     ,       AND STORE INDEX IN B REG
        035467  006016           ,LDAE    ,CMT,2  LOAD A REG WITH PARM PTR
        035470  033456
        035471  001010           ,JAZ     ,BANG   IF A=0 THEN ILLEGAL COMMAND
        035472  035567
 ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 41

        035473  054032           ,STA     ,PADR   ELSE STORE POINTER TO GET PARMS
        035474  006010           ,LDAI    ,' '    OUTPUT SEPARATOR BLANK
        035475  000240
        035476  002000           ,CALL    ,OUT    OUTPUT BLANK
        035477  033345
        035500  005002  GPRM     ,TZB     ,       CLEAR B REG AND
        035501  064023           ,STB     ,PLDR   PARM STORAGE INDEX
        035502  006010           ,LDAI    ,GP8    LOAD EXPR WITH DUMMY
        035503  035510
        035504  006057           ,STAE    ,EXPR   ERROR RTN ADDRESS
        035505  036255
        035506  002000           ,CALL*   ,PADR   GET 1ST PARM VALUE
        035507  135526
                035510  GP8      ,EQU     ,*
        035510  034014           ,LDX     ,PLDR   LOAD X REG WITH PSI
        035511  006055           ,STAE    ,P1,1   STORE RESULT IN PARM
        035512  035527
        035513  044011           ,INR     ,PLDR   INCR PARM STORAGE INDEX
        035514  044011           ,INR     ,PADR   POINT TO NEXT PADR LOC
        035515  001000           ,JMP     ,GPRM+2 PROCESS NEXT PARM
        035516  035502
        035517  002000  GMON     ,CALL    ,OFF
        035520  033132
        035521  002000           ,CALL*   ,MON    TO MONITOR
        035522  177540
                        *        COME HERE IF P AT MONITOR LEVEL
        035523  001000           ,JMP     ,GCOM+1
        035524  035417
                        *
                        *SPECIAL DATA AREAS
                        *
        035525  000000  PLDR     ,DATA    ,0
        035526  000000  PADR     ,DATA    ,0
        035527  000000  P1       ,DATA    ,0      PARAMETER STORAGE AREAS
        035530  000000  P2       ,DATA    ,0
        035531  000000  P3       ,DATA    ,0
        035532  000000  P4       ,DATA    ,0
        035533  000000  COM      ,DATA    ,0      COMMAND STORAGE AREA

 ISA220.DAS  DAS-10.8  [UCI 2-JUN-75] 18:39 23-AUG-75  PAGE 42

                        *
```

Appendix V

DECOMPILED ISADORA TARGET PROGRAM

This appendix contains the decompiled MOL620 program generated from the ISADORA source program listed in Appendix IV. The first two pages are the built-in procedures referenced by the decompiled code and are appended to the front of the decompiled text for recompilation.

This decompiled program corresponds to the P2 version of ISADORA described in Chapter 6. The only changes made to the P1 version (direct from the decompiler) are those needed to make the program compile and execute correctly. These changes are tabulated in Table 6-A.

Many labels appearing in this program are unused and could be removed if desired. These labels are underlined in the listing.

See Appendix III for a summary of the MOL620 language.

```
%       BUILT-IN DEFINITIONS AND PROCEDURES USED BY THE
        DECOMPILER GENERATED CODE       %

DECLARE AREG,BREG,XREG,OFLOW,T1,T2,T3 ;

EQU     TRUE=1,FALSE=0;          %TRUTH VALUES%

PROC EXC ;                  %EXTERNAL CONTROL
                             AR = FUNCTION CODE
                             BR = DEVICE ADDRESS    %
        ",LRLA,6" ;         %MOVE FUNCTION CODE%
        ",MERGE,031" ;      %A_A,B   FOR DEV ADDR%
        ",ORAI,0100000" ;         %EXC OPCODE%
        ",STA,*+1" ;        %STORE INSTRUCTION%
        ",EXC,0";           %EXECUTE IT BY FALLING INTO IT%
ENDP;

PROC OUTPUT ;               %OUTPUT TO DEVICE BR THE VALUE OF AR%
        ",STA,OUT$" ;
        ",LDAI,0103100" ;         %OUTPUT INST%
        ",MERGE,031" ;      %PUT IN DEVICE%
        ",STA,*+2" ;        %MAKE INSTRUCTION%
        ",LDA,OUT$" ;       %GET VALUE%
        ",OAR,0" ;          %OUTPUT INSTR EXECUTED HERE%
ENDP;
DECLARE "OUT$" ;            %LOCAL TO OUTPUT%

PROC INPUT ;                %INPUT TO AR FROM DEVICE AR%
        ",ORAI,0102500" ;
        ",STA,*+1" ;
        ",CIA,0" ;          %INSTRUCTION MADE HERE%
ENDP;

PROC SHIFT ;                %AR HAS THE ACTUAL SHIFT INSTRUCTION IN IT;
                             THE DATA TO BE SHIFTED IS IN VARIABLES
                             AREG AND BREG; RESULTS RETURNED IN SAME%
        ",STA,*+3";         %STORE SHIFT INST TO BE EXECUTED%
        ",LDA,AREG" ;
        ",LDB,BREG" ;       %GET DATA%
        ",NOP," ;           %SHIFT HERE%
        ",STA,AREG" ;
        ",STB,BREG" ;       %RESULTS BACK%
ENDP;

PROC SENSE ;                %SENSE DEVICE
                             AR = FUNCTION CODE
                             BR = DEVICE ADDR
                             RETURNS TRUE OR FALSE DEPENDING ON RESULT OF
                             SENSE  %
        ",LRLA,6" ;         %FN CODE%
        ",MERGE,031";       %OR IN DEV ADDR%
        ",ORAI,0101000" ;         %SENSE OP%
        ",STA,*+2" ;
        ",INCR,1" ;         %TRUE%
        ",SEN,0,*+3" ;      %SENSE HERE%
        ",TZA," ;           %FALSE%
ENDP;

PROC MUL ;                  %MULTIPLY AREG,BREG BY AR%
        ",STA,*+4";         %STORE DATA%
        ",LDA,AREG" ;
        ",LDB,BREG" ;
        ",MULI,0" ;         %MULTIPLIER IN SECOND WORD OF INST%
        ",STA,AREG" ;
        ",STB,BREG" ;
ENDP;

PROC DIV ;               %DIVIDE AREG,BREG BY AR%
        ",STA,*+4";      %STORE DATA%
        ",LDA,AREG" ;
        ",LDB,BREG" ;
        ",DIVI,0" ;      %DIVISOR IN SECOND WORD OF INST%
        ",STA,AREG" ;
        ",STB,BREG" ;
ENDP;


PROC AND ();                             %RETURN LOGICAL AND OF AR,BR%
        ",JAZ*,AND" ;                    % AR IS FALSE %
        ",TBA," ;                        % RESULT IS BR %
ENDP;

PROC "$SAVE" (AREG,BREG,XREG) ;          %SAVE PHYSICAL MACHINE REGISTERS
                                          IN VIRTUAL REGISTERS%
        OFLOW := (OF) ;
        (AR) := AREG ;
ENDP;

PROC "$LOAD" ;                   %MOVE VIRTUAL REGISTERS TO PHYSICAL REGS%
        (OF) := OFLOW ;
        RETURN (AREG,BREG,XREG) ;
ENDP;

PROC "$$SAVE" ( "$AR$", "$BR$", "$XR$") ;        %SAVE PHYSICAL REGS
                                                                 LOCALLY%
        "$OF$" := (OF) ;
        (AR) := "$AR$" ;                 %RESTORE A%
ENDP;

PROC "$$LOAD" ;                                  %LOAD PHYSICAL REGS%
        (OF) := "$OF$" ;
        RETURN ("$AR$", "$BR$", "$XR$") ;
ENDP;

DECLARE "$AR$", "$BR$", "$XR$", "$OF$" ;

%       END OF BUILT-IN STUFF   %
```

```
%
        THIS FILE FROM DECOMPILER WITH
        STAGE-TWO MANUAL FIXUPS (SYNTAX, SELF MODIFYING, DECOMP ERRORS) %

EQU     ZERO=0 , ONE=1 ,          %SMALL OFFSETS NEEDED%
        TWO=2 ;

EQU     L001=050 ;                %LOCATION (L001) WHERE TO SAVE PIM MASK%


% ISADORA DEBUGGER -- ICS DEPT -- UC IRVINE %
% VERSION 2.20 %
% ORIGINAL AUTHOR: MICHAEL PEPPER %
% ICS DEPT, UCI, 1969 %

% LAST UPDATE: G.L. HOPWOOD %
% 10-JUL-75 %

EQU BGNG = 033000 ;
EQU LO = 0200 ;
% LO,SET,END %
% HI,EQU,037777 UPPER BOUND OF USER AREA %
EQU HI = "BGNG-1" ;                       % UPPER BOUND OF USER AREA %
EQU BLD1 = 077630 ;                       % ADDR OF START OF LOAD ROUTINE %
EQU BLD2 = 077434 ;                       % ADDR OF START OF DUMP ROUTINE %
EQU MON = "077520+16" ;
EQU PIM = 0 ;
EQU PMLA = 1 ;
EQU PMLB = 6 ;
EQU MSKA = 0375 ;
EQU MSKB = 0277 ;
EQU IDVA = 4 ;                            % SERIAL CONTROLLER INPUT IF SS3 IS
                                                                      OFF. %
EQU IDVB = 1 ;                            % SERIAL CONTROLLER INPUT IF SS3 IS
                                                                       ON. %
EQU ODVA = 4 ;                            % SERIAL CONTROLLER OUTOUT IF SS3
                                                                 IS OFF. %
EQU ODVB = 1 ;                            % SERIAL CONTROLLER OUTPUT IF SS3
                                                                  IS ON. %
EQU ODV = 0 ;                             % SERIAL CONTROLLER OUTPUT FOR 611
                                                                   SCOPE %
% USER SHOULD SPECIFY THE FOLLOWING PARAMETERS BEFORE ASSEMBLY: %
% BGNG ORIGIN POINT FOR ISADORA %
% PIM PIM GROUP NUMBER (0,1,2) %
% PMLA INPUT PIM LINE FOR INPUT DEVICE IF SS3 IS OFF. %
% PMLB INPUT PIM LINE FOR INPUT DEVICE IF SS3 IS ON. %
% MSKA PIM MASK IF SS3 IS OFF. %
% MSKB PIM MASK IF SS3 IS ON. %
% IDVA INPUT DEVICE NUMBER IF SS3 IS OFF. %
% IDVB INPUT DEVICE NUMBER IF SS3 IS ON. %
% ODVA OUTPUT DEVICE NUMBER IF SS3 IS OFF. %
% ODVB OUTPUT DEVICE NUMBER IF SS3 IS ON. %
% C611 ADDRESS OF 611 CHARACTER OUTPUT ROUTINE %
% IF HE IS OUTPUTTING TO TEK SCOPE %
% LO LOWER BOUND OF USER WORK AREA %
% HI UPPER BOUND OF USER WORK AREA %
% BLD1 LOCATION OF START OF LOAD ROUTINE %
% BLD2 LOCATION OF START OF DUMP ROUTINE %

% DEBUGGING PACKAGE - INITIALIZATION ROUTINE %
% ROUTINE ENABLES AND OUTPUTS MASKS FOR ALL PIMS %
% THERE ARE THREE INTERRUPT SERVICE ROUTINES: %
% INA -- INTERRUPTS WHILE RUNNING DEBUGGER COME HERE %
% INP -- INTERRUPTS WHILE RUNNING USER PGM " %
% ANP3 - INTERRUPTS WHILE RUNNING ALL-NOT FUNCTION %

"*,ORG,BGNG" ;                              % ORG AT SPECIFIED LOC %
SET PMIA = "020*PIM+060" ;
SET PMIA = "2*PMLA+PMIA" ;                  % PIM INT LOCATION IF SS3 IS OFF. %
SET PMIB = "020*PIM+060" ;
SET PMIB = "2*PMLB+PMIB" ;                  % PIM INT LOCATION IF SS3 IS ON. %



DECLARE VERSN = "'2.20'" ;                  % IN CORE VERSION NUMBER %
PROCEDURE START %SYNTHETIC% ; % ADDRESS 33000 %
        AREG := -1 ;                        % A=-1 %
        IF NOT (SS3) % INITIALIZE TTY % THEN AREG := 0 ; % EXCHANGE SWITCH
                                                                          %
DFLT :                                      % USING SENSE SWITCH 3. %
        TSWIT := AREG ;
        CALL TTYEX ;                        % SET UP I/O FOR PROPER DEVICE %
        AREG := BREG := XREG := (OF) := 0 ; % ZERO ALL REGS AND CLEAR %
        CALL SAVE ;                         % REGISTERS %
        CALL GCOM ;                         % GO TO DEBUGGER %
ENDP;


% TELETYPE EXCHANGE ROUTINE: CHANGES I/O DEVICE TO %
% OTHER TTY THAN ONE SELECTED. %


PROCEDURE TTYEX ;                           % ADDRESS 33021 %
        IF (TSWIT := AREG := TSWIT BXR 0177777) = 0
                                            % IF THE TTY %
                                            % TRANSFER %
                                            % SWITCH IS %
                                            % A -1 THEN %
        THEN
          BEGIN
            PMIB := 02000 ;                 % SET UP INT LOC %
            "PMIB+1" := @INA ;
            AMSK := @MSKB ;
            CALL EXC (4, 040) ;
            L001 := L001 BOR @"0377-MSKA" ; % MASK OTHERS INT %
            CALL EXC (2, 040) ;
            OUTDEV := @ODVB ;
            INDEV := @IDVB ;
          END;
        ELSE
          BEGIN
            PMIA := 02000 ;                 % SET UP INTERRUPT AREA %
            "PMIA+1" := @INA ;
            AMSK := @MSKA ;                 % SET APROPRIATE MASK %
            CALL EXC (4, 040) ;             % DISABLE %
            L001 := L001 BOR @"0377-MSKB" ; % OR IN OTHER'S BIT %
            CALL EXC (2, 040) ;
            OUTDEV := @ODVA ;               % OUTPUT DEVICE %
            INDEV := @IDVA ;                % INPUT DEVICE %
          END;
%...THIS SECTION TAKEN OUT SINCE IT WAS MODIFYING PGM
TTY2 :                                      ### CHANGE: OUTPUT SENSE
                                                              INSTRUCTION ###
        OAR2 :=
        OAR1 := AREG := (SENO := AREG := AREG + 0101100) + 02000 ;
```

```
                                                ### OUTPUT CHAR INSTRUCTION ###
        CIA3 :=
        CIA2 := CIA1 := AREG := BREG + 0102500 ;
                                                ### GET INPUT DEVICE NUMBER ###
                                                ### AND ALL INPUT CHAR INSTRUCTIONS
                                                                               ###

        RETURN ;                                % RETURN %
ENDP;


PROCEDURE ON ;                                  % ADDRESS 33117. TURN ON OUR
                                                                  INTERRUPT %
        CALL EXC (4, 040) ;                     % DISABLE %
        TMP := AREG ;
        CALL OUTPUT (L001 := AREG := L001 BAND AMSK, 040) ; % GET MASK. OUR
                                                                      MASK %
        AREG := TMP ;
        CALL EXC (2, 040) ;                     % ENABLE PIM %
        RETURN ;                                % RETURN %
ENDP;


% OTHERS REMAIN ON %
PROCEDURE OFF ;                                 % ADDRESS 33133. TURN OFF OUR
                                                                 INTERRUPTS %
        CALL EXC (4, 040) ;                     % DISABLE %
        TMP := AREG ;
        CALL OUTPUT (L001 := AREG := (AMSK BXR 0177777) BOR L001, 040) ; %
                                                            COMPL OF OUR MASK %
        AREG := TMP ;
        CALL EXC (2, 040) ;                     % ENABLE PIM %
        RETURN ;
ENDP;


PROCEDURE WEC ;                                 % ADDRESS 33146. CHANGE INT VECTOR
                                                               ADDR TO B-REG VALUE %
        IF (AREG := TSWIT) = 0
        THEN
          BEGIN
WEC2 :      % DEVICE B % "PMIB+1" := BREG ; RETURN ; % DONE %
          END;
        "PMIA+1" := BREG ;                      % DEVICE A %
        RETURN ;
ENDP;


DECLARE TSWIT = 0 ;                             % TTY TRANSFER SWITCH %
DECLARE AMSK = MSKA ;                           % ACTUAL PIM MASK %
DECLARE TMP = 0 ;

% COMMAND T %
% ALLOWS USER TO CHANGE I/O DEVICE DURING EXECUTION %


PROCEDURE TTY ;                                 % ADDRESS 33163 %
        CALL TTYEX ; % CALL TELETYPE EXCHANGER % CALL GCOM ; % RETURN %
ENDP;
```

```
% INPUT CONTROLLER AND I/O SUBROUTINES %
% TTY INTERRUPTS COME HERE WHEN RUNNING USER PROGRAM %


PROCEDURE INP ;                          % ADDRESS 33170. I/O CONTROLLER %
        CALL "$SAVE" ;                   %MANUAL INSERT. SAVE REGISTERS%
        A := AREG ;                      % SAVE PRES A VALUE %
CIA1 :                                   % INPUT FROM DEVICE IDVA INITIALLY
                                                                         %
        AREG := INPUT (INDEV) ;
        CALL OFF ;
        AREG := AREG BOR 0200 ;          % PUT IN PARITY BIT %
        CALL LCASE ;                     % CHECK FOR LOWER CASE %
        IF AREG = ZI
                                         % IF CONTROL-N THEN %
                                         % PREPARE TO ENTER DEBUGGER %

        THEN
        IF INP < ZH
                                         % GET ADDR WHERE INTERRUPTED %
                                         % WAS IT IN OUR AREA? %
                                         % NO, JMP TO RETURN %

        THEN
          BEGIN
                                         % INTERRUPT WAS FROM OUR AREA, GET
                                                         SET FOR DEBUGGER %
            AREG := A ;                  % SET VARIABLES IN TABLE TO %
            CALL SAVE ;                  % USER'S VALUES. %
            BREG := @INA ;
            CALL WEC ;                   % RESET INT VECTOR %
            ZP := AREG := INP ;
            CALL CRLF ;                  % OUTPUT <*> AND GO TO DEBUGGER %
            AREG := '<*' ;
            CALL COUT ;
            AREG := '>' ;
            CALL OUT ;
            CALL GCOM ;                  % TO DEBUGGER, NO RETURN %
          END;
        % RESTORE AND RETURN %
DBG2 :
        AREG := A ;
        CALL ON ;                        % OUR INTS ON %
        CALL "$LOAD" ;                   %MANUAL INSERT. LOAD
                                          PHYSICAL REGISTERS%
        RETURN ;                         % RETURN TO USER'S PROGRAM %
ENDP;


% CHECK A-REG FOR LOWER CASE LETTER AND CHANGE TO %
% UPPER CASE IF NECESSARY. JUNE 30 1971. GLH %


PROCEDURE LCASE ;                        % ADDRESS 33247 %
        IF (AREG := AREG - 0341) < 0 THEN AREG := AREG + 0341 ; ELSE AREG
                                                          := AREG + 0301 ;
L002 :
        RETURN ;
ENDP;


% IN CONTAINS THE IDLE LOOP FOR DEBUGGER INPUT %
% WHEN USER IS TALKING DIRECTLY TO ISADORA. %
% ROUTINE 'INA' GETS THE INTERRUPT AND STORES THE %
% CHARACTER IN 'CHAR'. %
```

```
PROCEDURE IN ;                          % ADDRESS 33264. INPUT CHAR FOR
                                                                DEBUGGER %
        CHAR := AREG := 0 ;
        CALL ON ;                       % ENABLE OUR INT %
L003 :
        WHI~~LE~~ @TRUE
        DO
          BEGIN
            ",NOP," ;
            ",NOP," ;                   % IDLE LOOP FOR INTRPT %
            IF (AREG := CHAR) # 0
                                        % IDLE %
            THEN
              BEGIN
OAR2 :                                  % ECHO %
                CALL OUTPUT (AREG, OUTDEV) ;
                IF (AREG := AREG - 0212) = 0
                                        % LF %
                THEN
                  BEGIN
IN2 :               % ECHO CR CR WITH LF % AREG := 0106615 ; CALL COUT ;
                  END;
                ELSE
                IF AREG = 3
                                        % CR %
                THEN
                  BEGIN
IN4 :               % ECHO LF WITH CR % AREG := 0212 ; CALL OUT ;
                  END;
IN6 :                                   % ORIGINAL CHAR %
                AREG := CHAR ;
                RETURN ;                % RETURN %
              END;
          END;
ENDP;


% DEBUGGER INTERRUPTS NORMALLY COME HERE %
% WHILE ISADORA IS RUNNING %

PROCEDURE INA ;                         % ADDRESS 33326 %
        CALL "$$SAVE" ;                 %MANUAL INSERT. SAVE REGISTERS%
        A := AREG ;                     % SAVE AREG %
CIA2 :                                  % INPUT FROM DEVICE IDVA INITIALLY
                                                                        %
        AREG := INPUT (INDEV) ;
        CALL OFF ;                      % OUR INTS OFF, OTHERS ON %
        AREG := AREG BOR 0200 ;         % OR IN PARITY BIT %
        CALL LCASE ;                    % CHECK FOR LOWER CASE LETTER %
        CHAR := AREG ;
        AREG := A ;                     % RESTORE AREG %
        CALL "$$LOAD" ;                 %MANUAL INSERT. RESTORE REGS%
        RETURN ;                        % RETURN %
ENDP;

DECLARE CHAR = 0 ; DECLARE A = 0 ;      % SAVE AREA FOR A REG %

PROCEDURE OUT ;                         % ADDRESS 33345. OUTPUT CHAR FOR
                                                                DEBUGGER %
SENO :                                  % OUTPUT TO DEVICE ODVA INITIALLY %
        WHILE NOT (SENSE (1, OUTDEV)) DO ",NOP," ;
        IF (AREG := ((OUTA := AREG) BAND 0377) - 0212) # 0
                                        % LF? %
        THEN
        IF (AREG := AREG - @"0215-0212") # 0
                                                % CR %
        THEN
        IF AREG < @"´ ´-0215"
                                                % >=BLANK %
        THEN
          BEGIN
            AREG := ´?´ ; % MAKE A ? % GOTO OAR1 ;
          END;
OUT2 :                                          % RESTORE CHAR %
        AREG := OUTA ;
OAR1 :
        CALL OUTPUT (AREG, OUTDEV) ;
        AREG := OUTA ;                          % RESTORE AREG %
        RETURN ;
ENDP;

DECLARE OUTA = 0 ;                              % SAVE AREG %

DECLARE OUTDEV,                                 % OUTPUT DEVICE NUMBER %
        INDEV ;                                 % INPUT DEVICE NUMBER %


% PARAMETER TABLE %

EQU CMPT = "*" ;                                %PARAMETER TABLE%
DECLARE ZZDISP = (GLCA, GTYP, DSLR) ;           % < OR / %
DECLARE ZZEQ = (GVAR, GEXP, STVR) ;             % = %
DECLARE ZZSET = (GLCA, DSLR) ;                  % > %
DECLARE ZZQU = (GEXP, GTYP, DEXP) ;             % ? %
DECLARE ZZAN = (GLCE, GLCF, GEXP, GTYP, ALNT) ; % A,N %
DECLARE ZZB = (GNUM, GLCA, GEXP, BKPT) ; % B %
DECLARE ZZC = (GNUM, CLR) ;                     % C %
DECLARE ZZF = (GLCE, GLCF, GEXP, FILL) ; % F %
DECLARE ZZG = (GLCP, GO) ;                      % G %
DECLARE ZZR = PREG ;                            % R %
DECLARE ZZS = STEP ;                            % S %
DECLARE ZZCMNT = CMNT ;                         % : %
DECLARE ZZL = (GEXP, BLDE) ;                    % L %
DECLARE ZZD = (GLCE, GLCF, GEXP, BDMP) ; % D %
DECLARE ZZT = TTY ;                             % T %
DECLARE ZZM = (GLCE, GLCF, GLCE, MOVE) ; % M %


% COMMAND TABLE - HOLDS POINTERS TO CMPT %
% TO GET AN INDEX INTO THIS TABLE TAKE THE COMMAND CHARACTER %
% AND SUBTRACT ´:´. THE ENTRY AT THAT LOCATION IN THIS %
% TABLE POINTS TO THE PARAMETER TABLE ABOVE. %

DECLARE CMT = "(ZZCMNT)*" ;                     % : %
DECLARE * = 0 ;                                 % %
DECLARE * = "(ZZDISP)*" ;                       % < %
DECLARE * = "(ZZEQ)*" ;                         % = %
DECLARE * = "(ZZSET)*" ;                        % > %
DECLARE * = "(ZZQU)*" ;                         % ? %
DECLARE * = 0 ;                                 % @ %
DECLARE * = "(ZZAN)*" ;                         % A %
DECLARE * = "(ZZB)*" ;                          % B %
DECLARE * = "(ZZC)*" ;                          % C %
DECLARE * = "(ZZD)*" ;                          % D %
DECLARE * = 0 ;                                 % E %
DECLARE * = "(ZZF)*" ;                          % F %
DECLARE * = "(ZZG)*" ;                          % G %
DECLARE * = 0 ;                                 % H %
DECLARE * = 0 ;                                % I %
DECLARE * = 0 ;                                % J %
DECLARE * = 0 ;                                % K %
DECLARE * = "(ZZL)*" ;                         % L %
DECLARE * = "(ZZM)*" ;                         % M %
DECLARE * = "(ZZAN)*" ;                        % N %
DECLARE * = 0 ;                                % O %
DECLARE * = 0 ;                                % P %
DECLARE * = 0 ;                                % Q %
DECLARE * = "(ZZR)*" ;                         % R %
DECLARE * = "(ZZS)*" ;                         % S %
DECLARE * = "(ZZT)*" ;                         % T %
DECLARE * = 0 ;                                % U %


% ERRORS IN EXECUTION SUBROUTINES %


PROCEDURE LERR %SYNTHETIC% ; % ADDRESS 33512 %
DERR :  NULL ;
                                               % ADDRESS < LO %
        AREG := ´ ?´ ;
        CALL COUT ;
        AREG := ´<L´ ;
        CALL COUT ;
        GOTO BANG ;
ENDP;


PROCEDURE HERR %SYNTHETIC% ; % ADDRESS 33524 %
                                               % ADDRESS > HI %
        AREG := ´ ?´ ;
        CALL COUT ;
        AREG := ´>H´ ;
        CALL COUT ;
        GOTO BANG ;
ENDP;

% SPECIAL OUTPUT SUBROUTINES %

PROCEDURE CBK ;                                % ADDRESS 33536. OUTPUT A : %
        AREG := ´:´ ; CALL OUT ; RETURN ; % RETURN %
ENDP;


PROCEDURE CRLF ;                               % ADDRESS 33545. OUTPUT A CR/LF %
        AREG := 0106612 ; CALL COUT ; RETURN ; % RETURN %
ENDP;


% STEP - INSTRUCTION EXECUTION SUBROUTINE %
% STEP EXECUTES THE INSTRUCTION WHOSE 1ST WORD IS STORED IN %
% THE LOCATION GIVEN BY P. UPON EXECUTION THE B AND A REGS %
% CONTAIN THE 1ST AND 2ND WORDS OF THE INSTRUCTION(S) RESPEC- %
% TIVELY. %
PROCEDURE STEP ;                               % ADDRESS 33554 %
        BREG := ZERO [XREG := ZP] ;            % LOAD X REG WITH P. LOAD B REG
                                                                    WITH WORD AT P %
        CALL SHIFT (04151) % LSRB 9 % ; % CHECK FOR HALT INSTR %
        IF BREG = 0 % JMP IF HALT % THEN GOTO HALT ;
        BREG := ZERO [XREG] ;             % RELOAD %
        AREG := ONE [XREG] ;              % LOAD A REG WITH WORD AT P+ONE %
        CALL STPP ;                       % STEP %
        CALL GCOM ;                       % RETURN %
ENDP;


% STPP - STEP EXECUTION ROUTINE %
% STPP EXECUTES THE INSTRUCTION WHOSE 1ST WORD IS IN THE %
% B REG AND WHOSE 2ND WORD IF ANY IS IN THE A REG AS IF IT %
% WERE LOCATED AT P. SUBROUTINE IS USED BY STEP AND BREAKPOINT. %


PROCEDURE STPP ;                          % ADDRESS 33571 %
        SCND := "INST+1" := AREG ;        % SAVE INSTRUCTION DATA %
        % CHECK FOR SEN, IME, OME % AREG := INST := BREG ; % GET 1ST WORD
                                                                OF INST %
        CALL SHIFT (04346) % LSRA 6 % ; % LEFT 10 BITS %
        IF (AREG := (XREG := AREG) - 01020) # 0
                                          % SAVE %
                                          % IS IT 1020XX (IME) ? %
                                          % YES %
        THEN
        IF (AREG := AREG - 010) # 0
                                          % IS IT 1030XX (OME) ? %
                                          % YES %
        THEN
          BEGIN
            AREG := XREG ;                % GET BACK INST (LEFT 10 BITS) %
            CALL SHIFT (04343) % LSRA 3 % ; % LEFT 7 BITS NOW %
            IF (AREG := AREG - 0101) # 0
                                          % IS IT 101XXX (SEN) ? %
                                          % YES, TREAT LIKE JMP INST %
            THEN
              BEGIN
                % OTHERWISE CHECK SOME MORE TO SEPARATE INST TYPES % AREG
                                                                   := 0 ;
                CALL SHIFT (04444) % LLRL 4 % ;
                IF AREG = 0
                                          % DOUBLE WORD INSTRUCTION %
                THEN
                  BEGIN
                    % OP-CODE IS ZERO; TEST IF SINGLE NON-ADDRESS %
OPZ :
                    CALL SHIFT (04443) % NIL NIL % ;
                    IF AREG # 0
                    THEN
                      BEGIN
                        IF (AREG := AREG - THRE) < 0 THEN GOTO JMP ;
                        IF AREG = 0
                        THEN
                          BEGIN
                            % EXECUTE TYPE INCREMENT P FOR DOUBLE WORD %
EXEC :
                            BUMP ZP ;
                            CALL ADDR ; % GET OPERAND ADDRESS %
                            "INST+1" := AREG ; % STORE EFFECTIVE ADDR IN
                                                       2ND WORD OF INST %
                            BUMP STIF ; % NO CHECK FOR STORE, INR IMMED %
                            GOTO BAC2 ;
                          END;
                        IF (AREG := AREG - THRE) = 0
                        THEN
                          BEGIN
```

```
                                 % EXTENDED ADDRESSING; TEST IF RELATIVE %
EXT :
                                 BUMP ZP ;
                                 CALL SHIFT (04446) % LLRL 6 % ;
                                 AREG := 0 ;
                                 CALL SHIFT (04443) % LLRL 3 % ;
                                 IF (AREG := AREG - VIER) = 0
                                 THEN
                                   BEGIN
                                     % IS RELATIVE MAKE DIRECT %
EREL :
                                     INST := INST BOR 7 ;
                                     AREG := ("INST+1" + ZP) + 1 ;
                                     GOTO BAC1 ;
                                   END;
                                 GOTO BAC2 ;
                               END;
                           END;
                         GOTO BACK ;
                       END;
                     % TEST IF ADDRESS RELATIVE TO P % AREG := 0 ;
                     CALL SHIFT (04443) % LLRL 3 % ;
                     IF AREG = VIER
                                                 % IF A=0 THEN REL TO P %
                     THEN
                       BEGIN
                         % RELATIVE ADDRESS MAKE TWO WORD DIRECT %
REL :
                         "INST+1" := (INST BAND 0777) + ZP ;
                         BUMP "INST+1" ;
                         AREG := INST ;
                         CALL SHIFT (04351) % LSRA 9 % ;
                         INST := AREG := AREG BOR 06007 ;
                         GOTO BAC2 ;
                       END;
                     GOTO BACK ;
                     % SET NO-OP AS 2ND WORD %
BACK :
                     AREG := 05000 ;
                     GOTO BAC1 ;
                   END;
                 % JUMP TYPE TEST IF JUMP AND MARK %
JMP :
                 BUMP ZP ;
                 IF (AREG + 1) = 0
                 THEN
                   BEGIN
                     % SET RETURN FOR JUMP AND MARK %
JMPM :               AREG := @JMPS ; GOTO BAC1 ;
                   END;
                 % SET RETURN FOR JUMP % AREG := @JMPR ;
                 GOTO BAC1 ;
               END;
             % INST WAS IME OR OME, BUMP P-REG AND GO %
XME :
             BUMP ZP ;
             GOTO BAC2 ;
ENDP;

% RETURN FROM JUMP; PROCESS POSSIBLE INDIRECT %
PROCEDURE JMPR %SYNTHETIC% ; % ADDRESS 33720 %
        CALL ADDR ;
        ZP := AREG ;
        XREG := THRE ;                 % TEST TO MAKE SURE IT DOES %
```

```
L004 :                                          % JUMP INTO BKPT ROUTINE %
        WHILE (AREG := (BKRT [XREG]) - ZP) # 0
        DO
          BEGIN
          IF XREG = 0 THEN GOTO [STPP] ; XREG := XREG - 1 ;
          END;
JPBK :
        ZP := AREG := BLOC [CNT := XREG] ;
        CALL BKP ;
        GOTO [STPP] ;
ENDP;


% RETURN FROM JUMP AND MARK; PROCESS INDIRECT %
PROCEDURE JMPS %SYNTHETIC% ; % ADDRESS 33755 %
        CALL ADDR ; % STORE MARK IN JUMP ADDRESS SET P % XREG := AREG ;
        ZERO [XREG] := AREG := ZP + 1 ;
        ZP := XREG := XREG + 1 ;
        GOTO [STPP] ;
ENDP;

% GO THROUGH POSSIBLE INDIRECT ADDRESS CHAIN STARTING IN SCND. * LEAVE
                                                          RESULT IN A REG. %

PROCEDURE ADDR ;                                % ADDRESS 34016 %
        AREG := SCND ; ADRR : WHILE AREG < 0 DO AREG := ZERO [XREG := AREG
                                                              BAND 077777] ;
ENDP;

% DATA FOR STPP %
DECLARE SCND [1] ;
% STEP EXECUTION %
% STORE REVISED SECOND WORD %
PROCEDURE BAC1 %SYNTHETIC% ; % ADDRESS 34031 %
        "INST+1" := AREG ;
        ",NOP," ;
        % RESTORE REGISTERS %
BAC2 :  % LOAD ALL REGS AND OVFLW %
        CALL LOAD ;
        CALL "$LOAD" ;                          % RESTORE PHYSICAL REGISTERS %
        % STEP THROUGH INSTRUCTION %
INST :
        ",HLT,0" ;
        ",HLT,0" ;
        CALL "$SAVE" ;                          % SAVE PHYSICAL REGISTERS IN
                                                  VIRTUAL ONES %
        % SAVE REGISTERS; INCREMENT P; RETURN % CALL SAVE ; % SAVE ALL REGS
                                                                 AND OVFLW %
        BUMP ZP ;
        IF (AREG := STIF) = 0
                                                % IF IN BKP IGNORE TEST %
        THEN
          BEGIN
            CALL TSTI ;                         % TEST IF STI OR INRI %
            IF BREG = 0 THEN GOTO [STPP] ; % NO RETURN %
            ZERO [XREG := ZP - 1] :=
            AREG ;
                                                % ELSE STORE 2ND WORD OF INST %
                                                % BACK INTO USER'S PROGRAM %
            GOTO [STPP] ;                       % RETURN %
          END;
        GOTO [STPP] ;
ENDP;
```

```
                                        % TRIED TO STEP A HALT INSTR %
PROCEDURE HALT %SYNTHETIC% ; % ADDRESS 34063 %
        AREG := 'HA' ;
        CALL COUT ;                     % PRINT MSG %
        AREG := 'LT' ;
        CALL COUT ;
        CALL PREG ;                     % PRINT REGS, NO RETURN %
ENDP;



% TSTI - TEST IF A STORE OR INCR IMMEDIATE INSTRUCTION %

PROCEDURE TSTI ;                        % ADDRESS 34075 %
        AREG := INST ;                  % LOAD A REG WITH INST %
        INST := BREG := 0 ;             % CLEAR B REG AS FLAG. CLEAR INST
                                                     IN CASE OF 2ND PASS %
        IF (AREG := AREG - 06000) < 0
          THEN RETURN ;                 % IF INST<06000 THEN NOT. AN EXT OR
                                                                IMM INST %
        IF (AREG := AREG - 0100) >= 0
          THEN RETURN ;                 % IF INSTSTILL + THEN CANNOT BE. A
                                                     LOAD STORE INR INST %
        IF (AREG := AREG + 040) < 0
          THEN RETURN ;                 % IF INST<0 THEN INST IS A STORE.
                                                       OR INR EXT OR IMM %
        IF (AREG := AREG BAND VIER) = 0
                                        % IF 2 BIT ON THEN EXTENDED ELSE %
                                        % IMMEDIATE %
        THEN
          BEGIN
            AREG := "INST+1" ;          % IF IMM THEN LOAD A REG WITH %
            BREG := BREG + 1 ;          % INST OPER AND TURN ON FLAG %
            RETURN ;                    % RETURN %
          END;
        RETURN ;                        % EXTENDED RETURN %
ENDP;

DECLARE VIER = 4 ; % CONSTANT 4 % DECLARE THRE = 3 ; % CONSTANT 3 %


% EXECUTION SUBROUTINES %

% BLDE - LOAD A TAPE ON HIGH SPEED READER %
% COMMAND CHARACTER - L %
% SUBROUTINE LOADS A TAPE FROM THE HIGH SPEED READER %
% INTO CORE USING THE BINARY LOADER. %

PROCEDURE BLDE ;                        % ADDRESS 34133 %
        CALL BLD1 (0) ;                 % LOAD THE TAPE %
        CALL GCOM ;                     % RETURN TO GET NEW COMMAND %
ENDP;


% CMNT - COMMENT EXECUTION ROUTINE %
% COMMAND CHARACTER - : %
% ROUTINE ALLOWS ANYTHING TO BE TYPED AFTER COMMAND CHARACTER. %
% TO TERMINATE COMMENT PRESS CR. TO CONTINUE A COMMENT ONTO %
% NEXT LINE PRESS LF. %
```

```
PROCEDURE CMNT ;                               % ADDRESS 34141 %
L006 :                                         % INPUT COMMENT CHAR %
        WHILE @TRUE
        DO
          BEGIN
            CALL IN ;
            IF (AREG := AREG - 0212) = 0
              THEN CALL GCOM
            ;
                                               % IS IT A LF %
                                               % YES THEN END COMMENT %
            IF (AREG := AREG - 3) = 0
              THEN CALL GCOM
            ;
                                               % IS IT A CR %
                                               % YES THEN END COMMENT %
          END;
ENDP;


% PREG - PRINT ALL REGISTERS SUBROUTINE %
% COMMAND CHARACTER - R %
% SUBROUTINE PRINTS THE A B X AND P REGISTERS AND THE OVFLW IND. %


PROCEDURE PREG ;                               % ADDRESS 34156 %
        CALL CRLF ;                            % CR/LF BEFORE PRINTING %
        XREG := 3 ;                            % USE X REG AS POINTER %
L007 :                                         % TO SHOW REGISTER TO PRINT %
        WHILE @TRUE
        DO
          BEGIN
            AREG := RTAB [RIDX := XREG] ; % GET REG TO OUTPUT %
            CALL OUT ;                         % AND PRINT INDICATOR CHAR %
            RTYP := AREG ;                     % SAVE IND CHAR TO GET INDEX %
            CALL CBK ;                         % PRINT A : AND 2 BLKS %
            AREG :=
            Z [XREG := RTYP - '@'] ;
                                               % GET IND CHAR %
                                               % GET INDEX IN VAR TABLE %
                                               % SET X REG = INDEX %
                                               % SET A REG TO VAR DATA %
            CALL OTC ;                         % OUTPUT DATA IN OCTAL %
            AREG := TBK ;                      % OUTPUT 2 BLANKS FOR %
            CALL COUT ;                        % SPACING %
            IF (XREG := RIDX) = 0
                                               % LOAD X REG WITH IND PTR %
                                               % IF X=0 THEN PRINT OVFLW %
            THEN
              BEGIN
                AREG := 'O' ;                  % LOAD AND PRINT OVFLW %
                CALL OUT ;                     % IND CHAR %
                CALL CBK ;                     % PRINT : AND 2 BKS %
                IF (AREG := ZO) # 0
                                               % LOAD A REG WITH OVFLW %
                                               % IF O=0 THEN OVFLW OFF %
                  THEN AREG := 1 ;             % ELSE SET O=1 TO INDICATE %
L010 :
                AREG := AREG + '0' ;
                CALL OUT ;
                CALL GCOM ;                    % RETURN %
              END;
              ELSE XREG := XREG - 1 ;          % ELSE DEC PTR AND %
          END;
```

```
ENDP;

DECLARE RTAB = ("'P'", "'X'", "'B'", "'A'") ;
DECLARE RIDX = 3 ;
DECLARE RTYP = 0 ;


% DSLR - DISPLAY AND SET LOCATION SUBROUTINE %
% COMMAND CHARACTERS DISPLAY - < SET - > %
% SUBROUTINE PRINTS LOCATION VALUE AT LOCATION IF DISPLAY AND %
% WAITS FOR A VLUE TO BE SET TO THE LOCATION. TO DEFAULT A %
% BLANK FOLLOWED BY A CR OR LF IS NECESSARY. INPUTTING JUST %
% A CR OR LF DOES NOTHING TO THE SPECIFIED LOCATION. %

PROCEDURE DSLR ;                            % ADDRESS 34243 %
        BREG := 0 ;                         % CLEAR B REG TO IND FLAG %
        IF COM # '>'
                                            % LOAD A REG WITH < OR > %
                                            % SUBTRACT OFF A >' %
                                            % IF A=0 THEN SET ELSE %
          THEN BREG := BREG + 1 ;           % DISPLAY AND SET FLAG TO 1 %
L011 :                                      % SAVE FLAG FOR LATER %
        DSFG := BREG ;
        ZC :=
        [DSLC := Z := AREG := P1] ;
                                            % GET LOC PARM %
                                            % UPDATE @ %
                                            % STORE AS INIT LOC FOR DISPLAY %
DISP :                                      % LOAD A REG WITH H %
        WHILE (AREG := ZH - DSLC) >= 0
                                            % SUBTRACT OFF LOC PARM %
                                            % IF A<0 THEN H<S ERROR %
        DO
          BEGIN
            AREG := DSLC ;                  % LOAD A REG WITH LOC TO PRINT %
            CALL OTC ;                      % OUTPUT LOC IN OCTAL %
            CALL CBK ;                      % OUTPUT A : FOLLOWED 2 BKS %
            IF (BREG := DSFG) # 0
                                            % LOAD B REG WITH COM FLAG %
                                            % IF B=0 THEN SET AND SKIP DISPLAY
                                                                              %
            THEN
              BEGIN
                AREG := [DSLC] ;            % LOAD A REG WITH DATA AT LOC %
                CALL [P2] ;                 % AND OUTPUT IT IN SPECIFIED FORM %
                AREG := TBK ;               % SPACE 2 BLANKS %
                CALL COUT ;
              END;
SET :                                       % LOAD A REG WITH DATA AT LOC %
            SLOC := AREG := [DSLC] ;        % STORE AT SLOC AS DEFAULT ON SET %
L012 :                                      % CALL EXPR ANYZR TO PROCESS SET %
            WHILE
            @TRUE
            DO
              BEGIN
                CALL EXPR ;
                DSTP := AREG ;              % SAVE RESULT OF ANALYSIS %
                IF DEF >= 0
                                            % LOAD A REG WITH DEFAULT FLAG TO %
                                            % SEE IF ANYTHING WAS PROCESSED %
                  THEN SLOC := AREG := DSTP ; % IF NOTHING DONT SET LOC TO
                                            ANYTHING. ELSE SET LOC TO NEW DATA %
L013 :                                      % LOAD A REG WITH SADD INDEX %
                IF BREG < 9
```

```
                                            % DEC BY 9 TO SEE IF BK LF OR CR %
                                            % IF A< OR = 0 THEN CR OR LF OR ^ %
                  THEN
                    BEGIN
                                            % LOAD X REG WITH NEW LOC VALUE %
ROE :                 [DSLC] := XREG := SLOC ; % STORE IT AS NEW LOC VALUE %
                      IF (AREG := BREG - 7) < 0
                                            % SUB INDEX OF CRLF %
                                            % IF A<0 THEN CR DONE %
                      THEN
                        BEGIN
                                            % LOAD A REG WITH FINAL LOC %
DSFH :                    ZC := AREG := [Z := DSLC] ; % UPDATE @ WITH NEW LOC
                                                                              %
                          GOTO GC3 ;
                        END;
                      IF AREG = 0
                                            % LF %
                      THEN
                        BEGIN
ROE2 :                    BUMP DSLC ; % INCR TO NEW LOC % GOTO DISP ;
                        END;
                      IF (AREG := (DSLC := AREG := DSLC - 1) - ZL) < 0
                                            % MUST BE ^ %
                                            % GO BACKWARDS %
                                            % CHECK LOWER BOUND %
                                            % TOO LOW %
                        THEN GOTO LERR ;
                      GOTO DISP ;
                    END;
                  SLOC := AREG := 0 ;       % ELSE A BLANK SO ZERO A REG. AND
                                                 STORE AS DFAULT IN SLOC %
                END;
            END;
          GOTO HERR ;
ENDP;

DECLARE SLOC = 0 ;                          % NEW LOC DATA BUFFER %
DECLARE DSLC = 0 ;                          % LOC HOLDER %
DECLARE DSFG = 0 ;                          % FLAG TO SHOW COM 0=SET 1=DISPLAY
                                                                              %
DECLARE DSTP = 0 ;                          % TEMP DATA HOLDER %
DECLARE TBK = "'  '" ;                      % 2 BLANKS. USED FOR SPACING. %


% STVR - SET A VARIABLE %
% COMMAND CHARACTER - = %
% SUBROUTINE SETS A VARIABLE DESIGNATED BY THE INDEX IN %
% PARM 1 EQUAL TO THE VALUE IN PARM 2 %

PROCEDURE STVR ;                            % ADDRESS 34403 %
          Z [XREG := P1] :=
          AREG := P2 ;
                                            % LOAD A REG WITH NEW VAR DATA %
                                            % LOAD X REG WITH VARIABLE INDEX %
                                            % STORE NEW DATA IN VARIABLE %
          GOTO GC3 ;
ENDP;


% DEXP - DISPLAY A VALUE OF AN EXPRESSION %
% COMMAND CHARACTER - ? %
% COMMAND DISPLAYS THE VALUE OF AN EXPRESSION IN PARM2 IN THE %
% FORMAT SPECIFIED BY PARM 1. %
```

```
PROCEDURE DEXP ;                              % ADDRESS 34414 %
        AREG := P1 ;                          % LOAD A REG WITH DISPLAY VALUE %
        CALL [P2] ;                           % OUTPUT IN SPECIFIED TYPE %
        CALL GCOM ;                           % RETURN %
ENDP;


% BDMP - BINARY DUMP ROUTINE %
% COMMAND CHARACTER - D %
% ROUTINE PUNCHES AN OBJECT TAPE USING THE BINARY DUMP %
% ROUTINE. CONTENTS OF A B X REGS ARE SET BY THE %
% PARAMETER VALUES. %

PROCEDURE BDMP ;                              % ADDRESS 34423 %
        CALL GCOM ;                           % NO DUMP ANYMORE %
ENDP;

PROCEDURE L014 %SYNTHETIC% ; % ADDRESS 34426 %
        AREG := 0102204 ;                     % SUPPRESS PRINT TEMPORARILY %
        CALL COUT ;
        (OF) := 1 ;                           % SET OVFLW FOR PUNCH %
        CALL BLD2 (P1, P2, P3) ;              % BINARY DUMP %
        AREG := 0201 ;                        % TURN PRINT BACK ON %
        CALL OUT ;
        CALL GCOM ;                           % GET NEW COMMAND %
ENDP;


% A,N <START ADDR> <FINISH ADDR> <VALUE-LOOKING-FOR> <OUTPUT-TYPE> %
% ALNT - ALL AND NOT EXECUTION SUBROUTINES %
% COMMAND CHARACTERS ALL - A NOT - N %
% DISPLAYS ALL ADDRESSES & VALUES OF DATA THAT WHEN PARM 3 AND MASK M %
% ARE ANDED EQUAL TO THE DATA AT THE GIVEN ADDRESS. ERROR - %
% DATA AND M ARE ANDED SHOULD EQUAL PARM 3. ALL PRINTS ALL %
% OCCURRENCES OF EQUALITY AND NOT PRINTS ALL OCCURRENCES OF %
% INEQUALITY. %

PROCEDURE ALNT ;                              % ADDRESS 34451 %
        ACNT := AREG := 0 ;                   % ZERO COUNT %
        CALL CRLF ;                           % CR/LF BEFORE PRINTING %
        CALL PCHK ;                           % CHECK PARMS P1,P2 %
        BREG := @ANP3 ;                       % SET UP INT LOC %
        CALL WEC ;
        BREG := 0 ;                           % CLEAR B REG FOR FLAG %
        IF COM # 'N'
                                              % LOAD A REG WITH COM CHAR %
                                              % TO DETERMINE WHTHER ALL OR NOT %
                                              % IF A=0 THEN NOT ELSE %
          THEN BREG := BREG - 1 ;             % ALL. SET FLAG = -1 %
L015 :                                        % SAVE FLAG FOR LATER %
        ANFG := BREG ;
        ANLC := AREG := P1 ;                  % LOAD A REG WITH S(START LOC).
                                                                SAVE STARTING LOC %
        CALL ON ;                             % OUR INTS ON %
ANPS :                                        % LOAD B REG WITH COM FLAG %
        WHILE @TRUE
        DO
          BEGIN
            BREG := ANFG ;
            IF ([ANLC] BAND ZM) = P3
                                              % LOAD A REG WITH DATA AT LOC %
                                              % AND WITH MASK M %
```

```
                                         % SUBTRACT OFF VALUE %
                                         % IF A=0 THEN TEST IF ALL %
            THEN
            IF BREG = 0 % FOUND BUT NOT THEN SKIP PRINT % THEN GOTO ANP2 ;
                ELSE NULL ;
            ELSE
            IF BREG # 0 % IF NOT THEN PRINT LOC % THEN GOTO ANP2 ;
L016 :                                   % IF VALID FOR OUTPUT THEN %
            AREG := ANLC ;
            CALL OTC ;                   % LOAD A REG WITH LOC PRINT IN %
            CALL CBK ;                   % OUTPUT ´:  ´ %
            AREG := [ANLC] ;             % GET VALUE OF LOCATION %
            CALL [P4] ;                  % OUTPUT VALUE IN FORMAT %
            ACNT := (ACNT + 1) BAND 3 ;  % ACNT = NUMBER OF WDS ON LINE.
                                                                MODULO 4 %
            AREG := TBK ;                % 2 BLANKS %
            CALL COUT ;
            IF (AREG := ACNT) = 0 THEN CALL CRLF ; % 4 WORDS ON THIS LINE %
ANP2 :                                   % CHECK FOR TOO HIGH %
            IF (AREG := ANLC - P2) >= 0
                                         % DONE %
            THEN
              BEGIN
ANP4 :          BREG := @INA ;
                CALL WEC ;               % RESTORE REG INTERRUPT %
                CALL GCOM ;              % ENTER DEBUGGER %
              END;
              ELSE BUMP ANLC ;           % NEXT LOC %
          END;
ENDP;


PROCEDURE ANP3 ;                         % ADDRESS 34564. COME HERE ON INT
                                                 OR WHEN DONE WITH ALNT %
        CALL "$$SAVE" ;                  %SAVE PHYSICAL REGISTERS.MANUAL
                                                                  INSERT%
        AA := AREG ;                     % SAVE A %
        CALL OFF ;                       % OUR INTS OFF %
CIA3 :                                   % CLEAR INPUT REG %
        AREG := INPUT (INDEV) ;
        ANLC := P2 ;                     % LAST ADDR FOR SEARCH. CAUSE ALNT
                                                       TO STOP NEXT TIME %
        AREG := AA ;                     % RESTORE A %
        CALL "$$LOAD" ;                  %LOAD PHYSICAL REGISTERS%
        RETURN ;                         % RETURN %
ENDP;


DECLARE ACNT = 0 ;                       % COUNT # WORDS ON LINE %
DECLARE ANFG = 0 ;                       % COMMAND FLAG 0=NOT -1=ALL %
DECLARE ANLC = 0 ;                       % SEARCH LOC HOLDER %
DECLARE AA = 0 ;                         % SAVE A REG %

% PCHK -- UTILITY ROUTINE TO CHECK IF P1 <= P2. IF NOT, THEN %
% WE EXCHANGE P1 AND P2. %

PROCEDURE PCHK ;                         % ADDRESS 34603 %
        IF (AREG := P2 - P1) >= 0 THEN RETURN ; % OK %
        BREG := P1 ;
        P1 := AREG := P2 ;
        P2 := BREG ;                     % P1 AND P2 EXCHANGED %
        RETURN ;                         % RETURN %
```

```
ENDP;


% MOVE - M <START.SOURCE> <END.SOURCE> <START.TARGET> %
% MOVE WORDS [P1,P2] TO [P3,P3+P2-P1] %
% MUST HAVE P1<=P2, P2<=H, L<=P1, P3+P2-P1 <= H %

PROCEDURE MOVE ;                         % ADDRESS 34617 %
        CALL PCHK ;                      % CHECK PARMS P1,P2 %
        IF (AREG := (((P2 + P3) - P1) - ZH) - 1) >= 0
                                         % P3+P2-P1 (FINAL TARGET) > ZH %
          THEN GOTO HERR ;
        XREG := P1 ;                     % MOVE IT %
MV1 :
        WHILE @TRUE
        DO
          BEGIN
            IF (AREG := (XREG - P2) - 1) >= 0 THEN CALL GCOM ; % FINISHED %
            [P3] := AREG := ZERO [XREG] ; % SOURCE. TO TARGET %
            XREG := XREG + 1 ;
            BUMP P3 ;
          END;
ENDP;


% FILL - FILL LOCATIONS EXECUTION SUBROUTINE %
% F <START.LOC> <END.LOC> <VALUE> %
% SUBROUTINES FILLS LOCATIONS PARM 1 THROUGH PARM2 WITH THE %
% VALUE IN PARM 3. %

% WE USE THE MOVE ROUTINE %
% P1 := START (P1) %
% P2 := FINISH (P2) -1 %
% P3 := START (P1) +1 %
% [P1] := VALUE (P3) %
PROCEDURE FILL ;                         % ADDRESS 34650 %
        CALL PCHK ;                      % CHECK THE PARMS P1,P2 %
        P2 := P2 - 1 ;
        [P1] := P3 ;
        P3 := AREG := P1 + 1 ;
        CALL MOVE ;                      % NO RETURN %
ENDP;


% BKPT - SET BREAK POINT SUBROUTINE %
% SETS UP BREAKPOINT INFO. %
% BLOC HOLDS BREAKPOINT LOCATION. %
% BWD1 AND BWD2 HOLD INSTRUCTIONS REPLACED BY JUMP TO %
% BREAKPOINT PROCESSING. %
% BCNT HOLDS COUNT OF TIMES THROUGH BREAKPOINT. %
% BLMT HOLDS LIMIT ON TIMES THROUGH BREAKPOINT. %
% BTYP HOLDS FLAG ON TYPES OF INSTRUCTIONS REPLACED. %
% -1 = DOUBLE WORD INSTRUCTION %
% 0 = TWO SINGLE WORD INSTRUCTIONS %
% 1 = SINGLE FOLLOWED BY A DOUBLE WORD INSTRUCTION %
% CNT IIS USED ON ENTRY TO BREAKPOINT PROCESSING TO %
% INDICATE WHICH BREAKPOINT WAS ACTIVATED. %

PROCEDURE BKPT ;                         % ADDRESS 34665 %
        IF P1 >= 0
                                         % CHECK PARM 1 %
                                         % IF NEG, LIST BRKPTS %
            THEN
              BEGIN
                XREG := TRES ;                  % TEST TO MAKE SURE A %
                                                % BRKPT HAS NOT BEEN SET %
BKP2 :          WHILE (AREG := (BLOC [XREG]) - P2) # 0
                                                % AT THIS LOC PREVIOUSLY %
                                                % YES THEN ERROR %

                DO
                IF XREG = 0
                THEN
                  BEGIN
                    IF (AREG := BLOC [XREG := P1]) >= 0
                                                % TEST TO SEE IF PREVIOUS BKPT %
                                                % AT P1 %
                      THEN GOTO BERR ;
                    BLMT [XREG] := P3 ;
                    CNT := BCNT [XREG] := AREG := 0 ;
                    ZC := ZERO [Z := BREG := P2] ; % UPDATE @ %
                    BWD1 [XREG] := ZERO [BLOC [XREG] := BREG] ;
                    ZERO [BREG] := 01000 ;  % LOAD A REG WITH JMP INSTR %
                    BWD2 [XREG] := ONE [BREG] ;
                    ONE [BREG] := AREG := BKRT [XREG] ;
                    BREG := BWD1 [XREG] ;
                    % GET TYPE FLAG FOR 1ST WORD % CALL TYPA ;
                    IF (BTYP [XREG] := AREG) >= 0
                    THEN
                      BEGIN
                        BREG := BWD2 [XREG] ;
                        % GET TYPE FLAG FOR 2ND WORD % CALL TYPA ;
                        IF AREG # 0 THEN BTYP [XREG] := AREG := (AREG + 1) + 1
                                                                                ;
                      END;
                    % TERMINATE COMMAND AND RETURN %
BKPF :
                    CALL GCOM ;
                  END;
                  ELSE XREG := XREG - 1 ;
                GOTO BERR ;
              END;
            % OUTPUT LOCATIONS WITH BRKPTS ON THEM %
BKQ :       % START AT BRKPT 0 %
            XREG := 0 ;
L017 :                                          % GET LOCATION ADDRESS %
            WHILE @TRUE
            DO
              BEGIN
                IF (BLOC [XREG]) >= 0
                                                % NOT A BREAK PT %
                THEN
                  BEGIN
                    AREG := XREG + ' 0' ;
                    CALL COUT ;                 % OUTPUT #: %
                    CALL CBK ;
                    AREG := BLOC [BKQX := XREG] ; % SAVE X. GET BRK PT LOC %
                    CALL OTC ;                  % OUTPUT IT IN OCTAL %
                    XREG := BKQX ;              % RESTORE X %
                  END;
BKQ4 :                                          % NEXT %
                IF (AREG := (XREG := XREG + 1) - 4) >= 0
                                                % DONE? %
                                                % NO %
                  THEN CALL GCOM ;              % BACK TO TOP LEVEL %
              END;
ENDP;
```

```
DECLARE BKQX = 0 ;

% DATA FOR SETTING JUMP %
DECLARE BKRT = ("(BKO)", "(BK1)", "(BK2)", "(BK3)") ;


% TYPE ANALYSIS SUBROUTINE; 1ST WORD IN B REG; ON EXIT %
% A REG HOLDS -1 IF DOUBLE 0 IF SINGLE. %
PROCEDURE TYPA ;                          % ADDRESS 35030 %
        AREG := 0 ;
        CALL SHIFT (04444) % LLRL 4 % ;
        IF AREG = 0
        THEN
          BEGIN
            % OP ZERO IF M = 0 4 5 7 STILL SINGLE WORD %
TYPB :
            CALL SHIFT (04443) % NIL NIL % ;
            IF AREG = 0 THEN RETURN ;
            IF (AREG := AREG - TRES) = 0 THEN GOTO TYPD ;
            IF AREG < 0 THEN GOTO TYPD ;
            % M IS 4 OR GREATER; IF 6 SET DOUBLE WORD FLAG. %
TYPC :
            IF AREG = TRES THEN GOTO TYPD ;
          END;
TYPF :
        AREG := 0 ;
        RETURN ;
        % SET -1 AND EXIT %
TYPD :
        AREG := -1 ;
        RETURN ;
ENDP;

DECLARE TRES = 3 ;                        % CONSTANT 3 %


% EXECUTED BREAKPOINTS COME HERE %
% WE EXPECT CNT TO BE ZERO %
PROCEDURE BK3 %SYNTHETIC% ; % ADDRESS 35061 %
        BUMP CNT ;
BK2 :
        BUMP CNT ;
BK1 :
        BUMP CNT ;
        % SAVE REGISTERS %
BKO :   % SAVE THE REGISTERS %
        CALL "$SAVE" ;                    % SAVE PHYSICAL REGISTERS %
        CALL SAVE ;
        ZP := BLOC [XREG := CNT] ;
        IF (AREG := (BCNT [XREG]) - (BLMT [XREG])) >= 0
          THEN GOTO BKA ;
                                          % DONT BREAK; EXECUTE BREAKPOINT
                                                      INSTRUCTIONS %
        BCNT [XREG] := (BCNT [XREG]) + 1 ;
        BREG := @INP ;
        CALL WEC ;                        % SET UP INT LOC %
        CALL BKP ;
        CALL ON ;                         % OUR INTS ON %
        CALL "$LOAD" ;                    % LOAD PHYSICAL REGS %
        GOTO [ZP] ;                       % RETURN TO USER PGM %
ENDP;

% PROCESS BREAKPOINT; BK # IN X REG AND CNT %
```

```
PROCEDURE BKP ;                              % ADDRESS 35115 %
        STIF := 0 ;                          % CLEAR DOUBLE WORD FLAG %
        BREG := BWD1 [XREG] ;
        IF (AREG := BTYP [XREG]) < 0
        THEN
          BEGIN
            % DOUBLE WORD INSTRUCTION %
BDBL :
            AREG := BWD2 [XREG] ;
            BUMP STIF ;                      % TURN ON DOUBLE WORD FLAG %
          END;
        ELSE
        IF AREG = 0
        THEN
          BEGIN
            % SINGLE WORD INSTRUCTIONS %
BSNG :      CALL STPP ; BREG := BWD2 [XREG := CNT] ;
          END;
        ELSE
          BEGIN
            % SINGLE FOLLOWED BY DOUBLE; STEP THROUGH SINGLE % CALL STPP ;
            BREG := BWD2 [XREG := CNT] ;
            AREG := TWO [XREG := BLOC [XREG]] ;
          END;
BKB :
        CALL STPP ;
        IF (AREG := BTYP [XREG := CNT]) < 0
                                             % TEST IF DOUBLE WORD %
                                             % NO THEN SKIP CHECK %
        THEN
          BEGIN
            CALL TSTI ;                      % TEST TO SEE IF STI OR INRI %
            IF BREG # 0
                                             % IF NOT THEN CONTINUE PROCESSING %
              THEN BWD2 [XREG] := AREG ;    % ELSE UPDATE 2ND OF BKPT %
          END;
        % RESTORE REGISTERS AND GO %
BKD :   % CLEAR CNT AND DOUBLE WORD FLAG %
        CNT := STIF := AREG := 0 ;
        CALL LOAD ;                          % LOAD ALL REGS AND OVFLW %
        RETURN ;
ENDP;

% TAKE BREAK; RESET COUNT TO ZERO %
PROCEDURE BKA %SYNTHETIC% ; % ADDRESS 35200 %
        BCNT [XREG] := AREG := 0 ;
        CALL CRLF ;
                                             % PRINT BREAK POINT NUMBER THAT WAS
                                                                       EXECUTED %
                                             % E.G. <0> FOR BREAKPOINT ZERO. %
        AREG := '<' ;
        CALL OUT ;
        AREG := CNT BOR '0' ;
        CALL OUT ;
        AREG := '>' ;
        CALL OUT ;
        BREG := @INA ;
        CALL WEC ;                           % SET UP INT ADDR %
        CALL GCOM ;
ENDP;

% GO ROUTINE; CHECK IF BREAKPOINT %
```

```
PROCEDURE GO ;                                  % ADDRESS 35230 %
        CALL CRLF ;
        ZP := P1 ;
        XREG := 4 ;
BGOB :
        WHILE (AREG := (BLOC [XREG := XREG - 1]) - P1) # 0
        DO
        IF XREG = 0
        THEN
          BEGIN
            BKP := AREG := @GEND ;
            BREG := @INP ;                      % POINT INPUT TO INPUT CNTRLR %
            CALL WEC ;                          % INT LOC PTS TO INP %
            GOTO BKD ;
          END;
BGOA :
        CNT := XREG ;
        BREG := @INP ;
        CALL WEC ;
        CALL BKP ;
        CALL ON ;
        CALL "$LOAD" ;                          % LOAD PHYSICAL REGS %
        GOTO [ZP] ;                             % TO USER AGAIN %
ENDP;

PROCEDURE GEND %SYNTHETIC% ; % ADDRESS 35264 %
        CALL ON ;
        CALL "$LOAD" ;                          % LOAD PHYSICAL REGS %
        GOTO [ZP] ;
ENDP;

                                                % DATA STORAGE FOR BREAKPOINT %
        DECLARE CNT [1] ;
        DECLARE BTYP = (0, 0, 0, 0) ;
        DECLARE BLOC = ("-1", "-1", "-1", "-1") ;
        DECLARE BWD1 = (0, 0, 0, 0) ;
        DECLARE BWD2 = (0, 0, 0, 0) ;
        DECLARE BCNT = (0, 0, 0, 0) ;
        DECLARE BLMT = (0, 0, 0, 0) ;

        DECLARE STIF = 0 ;                      % DOUBLE WORD FLAG %


                                                % CLEAR BREAKPOINT SUBROUTINE %

PROCEDURE CLR ;                                 % ADDRESS 35335 %
        IF (AREG := P1) < 0
                                                % JMP IF WE CLEAR ALL %
        THEN
          BEGIN
CLR2 :      P1 := AREG := 0 ;
            CALL CLEAR ;                        % 0 %
            BUMP P1 ;
            CALL CLEAR ;                        % 1 %
            BUMP P1 ;
            CALL CLEAR ;                        % 2 %
            BUMP P1 ;
            CALL CLEAR ;                        % 3 %
            CALL GCOM ;                         % RETURN TO TOP LEVEL %
          END;
        ELSE
          BEGIN
            CALL CLEAR ; % CLEAR JUST ONE % CALL GCOM ; % NO RETURN %
          END;
```

```
ENDP;


PROCEDURE CLEAR ;                            % ADDRESS 35364 %
        IF (AREG := BLOC [XREG := P1]) < 0 THEN RETURN ; % RETURN, NO BKPT
                                                                    SET %
        BREG := AREG ;
        ZERO [BREG] := BWD1 [XREG] ;
        ONE [BREG] := BWD2 [XREG] ;
        BLOC [XREG] := AREG := -1 ;          % PUT -1 IN BLOC %
        RETURN ;                             % RETURN %
ENDP;


PROCEDURE BERR %SYNTHETIC% ; % ADDRESS 35406 %
        CALL CRLF ; AREG := 'PB' ; % PREVIOUS BKPT ERROR % CALL COUT ; GOTO
                                                                   BANG ;
ENDP;



% GCOM - COMMAND CONTROLLER AND PROCESSOR ROUTINE %
% SUBROUTINE INPUTS AND CHECKS COMMAND CHARACTERS. PROCESSING %
% OF THE COMMAND BY GETTING THE PROPER PARAMETERSOFF THE %
% COMMAND POINTER TABLES(CMT AND CMPT) AND BRANCHING TO THE %
% PROPER COMMAND EXECUTION SUBROUTINE. %

PROCEDURE GCOM ;                             % ADDRESS 35416 %
L020 :                                       % OUT INTS ON %
        WHILE @TRUE
        DO
          BEGIN
            CALL ON ;
            CALL CRLF ;                      % OUTPUT CR/LF TO TERMINATE
                                                     PREVIOUS COMMAND %
GC3 :         %INSERTED BY HAND; EXTERNAL ENTRY%
            (OF) :=
            PDEF := BREG := 0 ;
                                             % CLEAR B REG AND RESET %
                                             % PERMANENT DEFAULT FLAG %
                                             % RESET OVFLW IND IF IT WAS ON %
            AREG := '#' ;                    % PROMPT SIGN %
            CALL OUT ;                       % OUTPUT CHARACTER %
            CALL IN ;                        % INPUT COMMAND CHAR %
            IF (AREG := (COM := AREG) - 0203) = 0
                                             % SAVE COM CHAR FOR LATER %
                                             % IF CONTROL-C %
                                             % GO TO MON %
            THEN
              BEGIN
GMON :          CALL OFF ; CALL [MON] ; % TO MONITOR %
              END;
            ELSE
              BEGIN
                IF COM = '/'
                                             % RESTORE COMMAND CHARACTER %
                                             % / IS SYNONYM FOR < %
                THEN
                  BEGIN GC4: AREG := '<' ; END ;
                ELSE
                  BEGIN
                    AREG := COM ;
                    ",CALL,ILC,':','U'" ; % TEST FOR LEGAL COM CHAR %
                  T1 := (OF) ;          % IF NOT THEN ERROR %
                  (OF) := 0 ;
                  IF T1 THEN GOTO BANG ;
                END;
GC5 :                                   % ELSE GET AN INDEX ON CMT %
              IF (AREG := CMT [BREG := AREG - ':']) = 0
                                        % AND STORE INDEX IN B REG %
                                        % LOAD A REG WITH PARM PTR %
                                        % IF A=0 THEN ILLEGAL COMMAND %
                THEN GOTO BANG ;
              PADR := AREG ;            % ELSE STORE POINTER TO GET PARMS %
              AREG := ' ' ;             % OUTPUT SEPARATOR BLANK %
              CALL OUT ;                % OUTPUT BLANK %
GPRM :                                  % CLEAR B REG AND %
              PLDR := BREG := 0 ;       % PARM STORAGE INDEX %
L021 :                                  % LOAD EXPR WITH DUMMY %
              WHILE @TRUE
              DO
                BEGIN
                  EXPR := AREG := @GP8 ; % ERROR RTN ADDRESS %
                  CALL [PADR] ;         % GET 1ST PARM VALUE %
GP8 :                 %INSERTED BY HAND; EXTERNAL ENTRY%
                  P1 [XREG := PLDR] :=
                  AREG ;
                                        % LOAD X REG WITH PSI %
                                        % STORE RESULT IN PARM %
                  BUMP PLDR ;           % INCR PARM STORAGE INDEX %
                  BUMP PADR ;           % POINT TO NEXT PADR LOC %
                END;
            END;
          END;
ENDP;

% SPECIAL DATA AREAS %

DECLARE PLDR = 0 ;
DECLARE PADR = 0 ;
DECLARE P1 = 0 ;                        % PARAMETER STORAGE AREAS %
DECLARE P2 = 0 ;
DECLARE P3 = 0 ;
DECLARE P4 = 0 ;
DECLARE COM = 0 ;                       % COMMAND STORAGE AREA %
```