OBVIAS: A VISUAL INTERACTIVE EDITOR/ASSEMBLER ON THE CORVUS CONCEPT PERSONAL WORKSTATION

by

William Bruce Buzbee

B.S.J., University of Kansas, 1980

Submitted to the Department of Computer Science and faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

_____

Professor in charge

_____

_____

Committee members

_____

For the Department

_____

Date thesis accepted

Abstract

The primary task of the student in computer science is to learn. This task, however, is often undermined by inefficient programming environments - environments in which the student spends more time locating program errors than learning important concepts. There are also many concepts in computer science which are difficult to visualize and/or understand - such as the inner workings of a computer which is carrying out operations in the millionths of a second.

The purpose of this thesis is to describe the development of OBVIAS: Our Best Visual Interactive Assembler and Simulator - a software system designed to provide an efficient assembly language programming/learning environment. OBVIAS supports the Motorola MC68000 assembly language, and was developed for and on the Corvus Concept Personal Workstation. It utilizes the large display of that machine to present a visual model of the key data states of the machine and shows these changing as a program executes.

OBVIAS aids in debugging by providing a character-by-character syntax checking editor/assembler and a dynamic visual execution model. OBVIAS performs syntax checking and incremental assembly by utilizing the power of concurrent productions, and helps alleviate run-time and logic errors by simultaneously displaying the student's

source code and a dynamic model of the CPU during visual execution. Further, the learning process is accelerated by OBVIAS' comprehensive help system and general user-friendly attitude.

This thesis first presents the initial specifications for the system and the environment in which it is to be developed. After finalization of the specifications based on the existing hardware/software environment and the course related expectations, the implementation process itself is described. This includes the implementation strategy, the principle problems encountered, the solutions to those problems, the overall program structure and the data structures used.

The product is a complete software system, which should have substantial value both in a student learning and a program development setting.

Dedication

To my parents, who made it all possible.

Acknowledgements

Without question, the success of this project is due to the years of research and development work by Professor Earl J. Schweppe. For this, and his guidance and support, he has my gratitude.

Table of Contents

1. Introduction

A software development tool can take many forms, but its primary purpose is constant: to provide an environment in which a programmer can more efficiently produce software. Similarly, a software educational tool is designed to provide an environment in which a student can more efficiently learn. As this thesis will show, the goals of software development and educational tools need not be mutually exclusive. This thesis will be devoted to tracing the development and implementation of a software tool which fulfills the goals of both software development and educational tools, depending on the setting in which it is used.

In the summer of 1983, the University of Kansas acquired a network of eight Corvus Concept Personal Workstations, powerful MC68k based computers. These machines were to be used in teaching CS 400, Computer Systems and Concurrent Processes. The course introduces the physical organization of computers, featuring approximately 12 laboratory programming projects designed to give students practical experience in assembly language programming, and includes an introduction to operating systems and concurrent processes. The goal of the work leading up to this thesis was to produce a software tool which would provide a more efficient and friendly programming environment so that the students in the course

would spend less time trying to get their programs to work correctly, and more time learning the concepts behind the projects.

At the time this project began, the general procedure for completing one of the programming projects was as follows:

1. The student would study the project description until he or she understood what tasks the program was to perform.
2. The student would prepare an algorithm to solve the problem.
3. The student would convert the algorithm to MC68k assembly language.
4. The student would enter his or her program into the computer using a text editor.
5. The text file produced by the text editor would be processed by an assembler, which would convert the assembly language into object code. If, however, the student made any syntax errors in writing the assembly language code or in entering it into the computer, the assembler would abort, listing the errors.
6. If there were any errors detected during assembly, the student would have to find them, correct them using the text editor, and go back to step 5.
7. After successfully completing the assembly process,

the student would use a program called the linker to transform the object module produced by the assembler into executable code.

8. The student would then run his program and test it to ensure that it performed according to the specifications outlined in the project description.
9. If errors were detected during execution of the program, the student would have to manually trace the execution of the program on paper, find the error, correct it using the text editor and restart at step 5.

In practice, students tended to spend a disproportionate amount of time trying to correct errors - often to the exclusion of learning important material. Ideally, a software tool to remedy this situation should not only reduce time spent attempting to correct errors to an absolute minimum, but should also give the student a better insight into the program development process and the operation of the computer itself. The remainder of this thesis will show how this was accomplished.

The structure of this thesis will, in general, mirror the actual development process of the software tool. In section 2, the problems to be solved are explained in detail, along with the first stage solutions to them. Also included are features that would have been helpful or desirable, but were not or could not be implemented. This

section roughly corresponds to the software specification stage of program development.

Section 3 is devoted to a description of the actual computers that the system was developed for and on. This is particularly important in that the solutions to some of the problems were possible only by utilizing some of the more powerful features of this particular machine.

Section 4 details the final design stage of the tool, showing how the problems determined in section 2 are to be solved using the hardware described in section 3.

Section 5 concerns actual implementation of the program, and includes discussions on implementation trade-offs. This section will also discuss problems encountered during the implementation phase.

Closely related to section 5, section 6 details the data structures chosen for the implementation.

Finally, some conclusions to this effort are presented. A series of actual screen images of the finished system is included as Appendex I.

The basic texts for the research leading to this thesis were Aho and Ullman [I], Ghezzi and Jazayeri [G], and Hopcroft and Ullman [H].

2. Initial Specification Phase

The initial specification phase of a software system can take many forms, but it must do one thing: determine what the program is to accomplish. This turned out to be one of the more difficult tasks of the development process.

Much of the software that is developed today is an improvement on or modification of existing software, and thus its designers have the luxury of comparison during the initial design phase. For example, if one were designing a word processor, he or she would first become familiar with the best existing word processors and note the good and bad features of them before designing a new word processor. In this case, however, no existing system fully exploited the capabilities of the modern microcomputer in an assembly language programming environment. However, several existing systems include the foundation for a comprehensive assembly language programming environment. Two such systems were developed at the University of Kansas.

In 1973, a system similar to the one described herein, but very restricted in scope, was implemented on the Datapoint 2200 computer [D]. Designed by Dr. Schweppe and programmed by Paul F. Heubner and Daniel T. Skelton, the system accepted Datapoint 2200 mnemonics, performing character-by-character syntax checking and interactive assembly. Following entry, the code could then be visually emulated using a CPU model displayed on the CRT. This

program was limited in that labeled statements were not allowed and operands were restricted to literals. Research on this model was also conducted by Martha Lee [B].

A more ambitious model was partially implemented on a Terak computer at the University of Kansas in 1980. An expansion of the Datapoint 2200 model, it was designed by Dr. Schweppe and programmed by Eldon Roehl [K]. It too, featured character-by-character syntax checking and interactive assembly, this time for for a subset of PDP-11 assembly language.

The limited nature of the two earlier systems was primarily a result of the state of computer hardware at the time they were developed. Considering the machines they were developed on, both systems were exceptional. The new system, however, would have the benefit of a far more powerful machine. Its potential, therefore, was correspondingly high.

To determine what tasks the new system was to perform, it was first necessary to isolate the problems and weaknesses in the present system, and determine what could be done to correct them. Thus arose "SUPERTOOL," a set of specifications for an imaginary software system that would not only provide the students with all possible help, but some impossible help as well. This unrealistic specification would be used as a goal when drafting the true specifications. The theory was that by designing an

imaginary system that could do everything, the designers would be freed from a conventional mind-set in which new ground might be overlooked simply because it had never been done before.

Actually, this design process began more than a decade ago, and most of the initial specifications were worked out by Dr. Schweppe and several of his graduate students. These specifications have evolved as new computer capibilities developed.

The assumptions about the CS 400 laboratory were also gathered largely from Dr. Schweppe and the various student assistants who have managed the laboratory since the course began.

The course is not intended to be a general programming course, yet much of the time spent by students is devoted to working on their assembly language programming projects. The projects are designed to give the students practical experience with topics in machine organization. In practice, the students tended to spend a disproportionate amount of time locating syntax and logic errors in their projects, rather than concentrating on the purpose behind the projects. The reason for this is that the assembly language programming environment in the CS 400 laboratory is not very friendly - a common fault in most low-level programming installations. The first assumption, then, is that the students would be better served by an environment

in which their time would be spent more productively. Ideally, this environment would not only streamline the programming and debugging process, but would help students to better understand the entire process and overall machine organization.

Having determined that the environment needed to be improved, the next stage was to isolate the critical activities in which time was not being used efficiently. This was not difficult. The primary one was project debugging. In general, two types of errors are common in student projects - syntax and logic errors. Syntax errors are detected during the assembly process, and are most often the result of misspellings or an illegal application of a particular addressing mode. In the existing environment, there is some help for syntax errors. The assembler shows the line on which an error occurs, and displays a cryptic error message. With logic errors, however, the existing environment provides very little help. If a logic error results in a error trap, the system simply halts, offering absolutely no explanation. Once students work out their syntax errors with what little help the assembler gives, they are completely on their own. The only recourse available to them in locating logic or run time errors is to hand trace the execution of their code, clearly a time consuming and not particularly productive process.

The next most damaging activity was the process of learning the physical structure of the processor and memory. For many students, CS 400 provides the first glimpse of what is actually inside a computer, and many students struggle with the concepts throughout the course. This lack of understanding shows up in the form of programming logic errors, as well as incorrect answers on the examinations.

Finally, many students still have trouble grasping what is really happening during the program development process, especially when confronted with the detailed level at which one must work in assembly language. Once again, these misconceptions manifest themselves as programming errors.

In brief, the new system would have to streamline the programming process, either eliminating errors entirely or providing assistance to the students so that they could find and correct them. Further, it should aid in teaching several of the most important topics in the course. Those topics should include machine organization and the underlying processes involved in the program entry to program execution cycle.

The last assumption is perhaps the most important - that a picture is indeed worth a thousand words. In the normal programming environment a student sees the input, the program and the output only in a static form. On the

other hand, a student who is shown a dynamic visual model or demonstration of a process or concept during its execution will learn more quickly and understand more fully than a student who simply reads about it in a book or programs in a conventional environment.

Based on these assumptions, the initial specifications for SUPERTOOL were drafted. Those specifications follow:

1. Because one of the prime functions of CS 400 is to teach machine architecture utilizing the Motorola MC68000, SUPERTOOL should provide a visual model of the microprocessor, including a text tutorial and quizzes.
2. SUPERTOOL should provide a complete programming environment for the MC68k assembly language, including an editor, assembler, linker, relocating loader and visual execution model.
3. All modules within SUPERTOOL should be fully compatible with existing programs in the Corvus environment. For example, a text file produced by EdWord, Corvus' text editor, should be permitted to enter the SUPERTOOL environment. Likewise, an object module created by SUPERTOOL should be of the same format as the object modules required by the Corvus supplied linker.

4. Because its primary function is to teach, SUPERTOOL should be capable of visually demonstrating all relevant tasks during the program entry to program execution phase.
   a. During program entry, SUPERTOOL should actually assemble the program as it is entered, visually displaying the incremental assembly of the machine code as new information becomes available.
   b. If requested, SUPERTOOL should give a visual demonstration of symbol table search/entry operations using a user-defined algorithm (hash, linear, tree, etc.).
   c. SUPERTOOL should be capable of demonstrating the actual execution of the code line immediately after it is entered, if possible.
   d. During the linking process, SUPERTOOL should visually demonstrate the searching of object code libraries and the definition of external references.
   e. During the relocation/loading process, SUPERTOOL should visually demonstrate the relocation of the affected code lines.
   f. SUPERTOOL should provide a comprehensive visual model of the actual execution of the program.

g. All of the above visual specifications should be capable of being turned on or off, so that the user can focus on the topic currently being studied.

5. All manuals for SUPERTOOL should be built into the system. Further, these manuals should be retrievable in two forms.
   a. At any time, the user should be able to retrieve help on a specific subject by giving the system the name of the target subject.
   b. At any time, the user should be able to make a general request for help, in which case SUPERTOOL will determine what information is most likely needed based on what the user is currently doing.
6. Detailed prompting must exist throughout the system. At all times, all possible input choices should be displayed.
7. In keeping with SUPERTOOL's role as a teaching device, it must provide a tutorial on the syntax of MC68k assembly language.
8. SUPERTOOL must provide a tutorial on the function of all MC68k operation codes and pseudo operations.
9. SUPERTOOL must provide a tutorial on the function of all MC68k addressing modes.

10. SUPERTOOL must provide a tutorial on which addressing modes are legal with which opcodes.
11. SUPERTOOL should include a comprehensive visual tutorial on the development and efficiency of algorithms.
12. No programming errors of any kind are to be permitted.
    a. Syntax errors are simply not to be allowed during the entry/assembly phase.
    b. Logic errors (code which deviates from the algorithm produced by the algorithm tutorial) are to be detected and the user warned.
    c. Run time errors are to be detected and the user warned.
13. The editor shall format the source code as it is entered.
14. Because different users will have different levels of expertise, SUPERTOOL must be capable of configuring itself according the needs of each user. A beginner will get all the prompts, but an expert user will get fewer.
15. SUPERTOOL must be extremely easy to use and understand. It should not execute in "teletype" mode, but rather should present the user with an extremely visual interactive environment.

Clearly, the specifications for SUPERTOOL are ambitious, but they turned out to be extremely valuable. The final system achieved far more than was initially thought possible, largely because of the fact that many of SUPERTOOL's features were included in the actual initial design even though it was believed that they were too difficult to implement. Some of the nicest features of the completed system were not expected to survive the implementation phase. Had a truly "realistic" initial system been designed, those features probably would have been dropped at the start.

The next step in the initial design phase was to modify the SUPERTOOL specifications into a more realistic, but still ambitious, package. This modification would result in the specifications for the finished system, OBVIAS - Our Best Visual Interactive Assembler Simulator. The general strategy for this modification was to evaluate each specification for SUPERTOOL using the following questions:

1. Is it possible, given current software and hardware technology?
2. If the goal is not completely possible, how closely can a set of new specifications come to achieving it?
3. Is the specification within the scope of this project?

4. Does the benefit of implementing the specification justify its cost in terms of programmer time and effort?

By subjecting each of SUPERTOOL's specifications to this test, dropping and modifying specifications where necessary, the initial set of specifications for OBVIAS was drafted.

Specification 1, that the system should provide a complete visual model of the Motorola MC68000 microprocessor, passed the test on all counts. OBVIAS is based on such a visual model. The sub specifications, that the system should include text tutorial and quizzes, however, was determined not to be within the scope of this project. The function of the tutorial and quizzes, however, were not to be overlooked. Text explaining the MC68000 could be easily integrated into the comprehensive help system outlined in specification 5. A quizzing unit would probably be better implemented as a stand-alone package. A quizzing tutorial would not provide the desired continuity within the system, which was envisioned as a highly interactive visual environment in which the user is in control. In a quizzing tutorial, the machine is generally in control - asking questions and demanding answers.

Specification 2, that the system should provide a complete programming environment, survived in spirit, but was somewhat scaled down. First, it was decided that a relocating loader was unnecessary for two reasons. Most student projects naturally fall into the memory independent code category. Thus, a demonstration of relocation would occur very infrequently. Secondly the early conception of the system required that code be assembled directly into memory. Clearly, the editor and assembler were absolutely necessary, but, in line with specification 12b., that code be assembled and syntax checked during entry, the two could be combined into a single unit.

The final requirement, that the system include a linker, posed the most difficult decision. It would be beneficial in a programming environment for a user to be able to call upon libraries of object code. In the final decision, however, it was determined that it was not within the scope of the project. One of the primary benefits of the execution unit (as it was envisioned), was that the user could simultaneously see the text of his source code and its execution. With standard object files, the text would not be present. It was decided to simply "hard code" the most often used library routines into the system. In that way, their format could be readily controlled.

Specification 3, that the system should be fully compatible with existing Corvus software, was reduced in

some places, and expanded in others. It was decided that the new system need not actually produce object or load modules because they were unnecessary for it to function. The system, however, would be compatible in that its assembler would conform to the syntax of Corvus' assembler and the code text files would be interchangeable between OBVIAS' editor/assembler and Corvus' word processor. Thus, a user could develop a program using OBVIAS, save the code text file and then run it through the Corvus assembler, linker and loader if he or she wished it to function outside the OBVIAS environment. Further, if a user wished to bypass OBVIAS' syntax checking editor/assembler, he or she could use any standard ASCII text editor and then load it into the OBVIAS environment.

Compatibility, when speaking in terms of a software environment, was determined to be very important. For this reason, it was decided that the system should "appear" to the user as an extension of the Corvus supplied software. In other words, screen formats and command syntax would be patterned after Corvus' example whenever possible. In doing this, it was hoped that the user would more easily become accustomed to the OBVIAS environment.

Specification 4, that the system should provide a visual demonstration of all important tasks during the program development process, is one of the more ambitious requirements. Nevertheless, it is one of the most

important. This system was designed with the theory that the students would learn more in less time if they were given a visual demonstration of the topic. Thus, visual demonstrations were given high priority.

Specification 4a, that the system should provide a visual demonstration of the incremental assembly process, was adopted intact.

Specification 4b, that the system should provide a visual demonstration of symbol table search/entry, was dropped. This feature was possible and helpful, but determined not to be within the scope of this project. Such a teaching device would be better accomplished as a stand-alone unit.

Specification 4c, that the system should be capable of demonstrating the actual execution of the code line immediately after it is entered, was adopted intact. This feature was believed to be particularly useful for students attempting to learn the function of the MC68k instructions.

Specification 4d, that the system should visually demonstrate the linking process, was dropped because OBVIAS was not to include a linker.

Specification 4e, that the system should visually demonstrate the relocation/loading process, was also dropped because OBVIAS was not to include a relocating loader.

Specification 4f, that the system should provide a

comprehensive visual model of the actual execution of the program, was adopted intact. This would become one of the key features of the system.

Specification 4g, that all of the visual demonstrations be switchable, was conditionally adopted. The reason for switching off a visual display is generally that doing so will provide an improvement in execution speed or available display space. It was decided that if such a trade-off existed, a switch would be provided. But if nothing were to be gained by switching off a particular visual model, it should always be kept on.

Specifications 5, 5a and 5b were adopted intact. Users, particularly novice users, spend a considerable amount of time searching through users' manuals. By building the manuals into the environment and essentially "letting the computer turn the pages," the user would have more time to devote to the problem at hand. This specification was also somewhat broadened to include error messages. Much of the students' criticisms of the CS 400 programming process revolved around overly cryptic system error messages. It was decided to treat error messages in much the same manner as requests to the system for help. Instead of receiving a cryptic error message noting a problem, an OBVIAS user would, if he or she requested it, receive as full an explanation of the problem as possible, as well as suggestions for correction and recovery.

Specification 6, that detailed prompting should exist throughout the system, was adopted intact. It was determined, however, that the prompting should, whenever possible, be integrated into the expanded manual help features outlined in specification 5.

Specification 7, that the system should provide a tutorial on the syntax of MC68k assembly language was implicitly adopted in that the combination of the syntax checking editor/assembler, the help/prompt system, and the immediate code execution feature would provide an equivalent function. Similarly, specifications 8, 9 and 10 were implicitly adopted via the same reasoning.

Specification 11, that the system should provide a comprehensive visual tutorial on the development and efficiency of algorithms, was dropped. Such a system is clearly beyond the scope of this project.

Specification 12, that the system should permit no programming errors of any kind, was impossible to fully adopt. It was decided, though, to attempt to come as close as possible to this specification. Specification 12a, that the system not permit syntax errors during the entry/assemble phase, was fully adopted. Specification 12b was dropped not only because the algorithm module was not to be included, but because it too is beyond the scope of this project. Although the system itself would not detect logic errors, it was decided to emphasize the execution

module of the system so that the user would be presented with an extremely friendly and helpful environment to assist him or her in finding logic errors.

Finally, and in conjunction with the modified specification 12b, the run time error detection and recovery specification was fully adopted. It was decided that any run time error that could be detected, should be detected and fully explained.

Specification 13, that the editor should format the source code as it is entered, was fully adopted. Not only should this make the students' code more visually appealing, but it is intended to encourage them to always produce clean, well documented code on whatever future system they use.

Specification 14, that the system should be configurable, was conditionally adopted. It was determined that although it is possible to force the system to configure itself in response to assumptions about the current user, it would not be worth the effort. Also, in most cases, a user is more knowledgeable about his or her level of proficiency than the system could be.

The final specification, that the system should be friendly and easy to use, was whole heartedly adopted. Ease of use was to become a prime consideration throughout the development process.

To summarize, the initial design specifications for the final system, OBVIAS, follow:

1. OBVIAS shall provide a complete visual model of the Motorola MC68000 microprocessor.
2. OBVIAS shall provide a complete MC68k assembly language programming environment.
    a. The functions of a text editor and assembler will be combined into a single unit. Character by character syntax checking and incremental assembly will be performed during program entry.
    b. A visual execution module will be included to demonstrate the execution of a user entered program. This module will combine the visual model of the Motorola MC68000 microprocessor with the user's source code text in order to trace execution. As each line in the user's program is executed, it will be marked on the display, and the MC68000 visual model will be updated to show the execution results.
3. Whenever possible, OBVIAS shall attempt to be consistent with existing Corvus software.
    a. Source code text files used by the editor/assembler shall be compatible with text files produced by Corvus' word processor, Edword.

b. The command structure, screen formats and operation protocols used by OBVIAS shall be as consistent as possible with those used by Corvus software.

4. OBVIAS shall provide detailed visual demonstrations of selected operations in the code entry to program execution process.
   a. A visual demonstration of the incremental assembly shall be provided.
   b. The option to visually execute a code line immediately following entry shall be provided.
   c. A comprehensive visual execution model shall be provided.
   d. If the inclusion of any visual model presents a significant performance decline, the user will be given the option of bypassing it.
5. A comprehensive system to aid the user in utilizing OBVIAS shall be integrated into each applicable module.
   a. The OBVIAS user's manual and a MC68k programming manual shall be built into the system.
   b. At any time, a user may request information on a specific topic by informing OBVIAS of the target subject.
   c. At any time, a user may request general help.

OBVIAS will then select what information is most likely needed based on what the user is doing.

d. At all times, the user will be presented a prompt informing him or her what command selections or code entry choices are permissible.

e. The error message system shall be functionally handled as part of the overall help system. When a user's action generates a system error, the user will be given the option of viewing a detailed description of that particular error. This description shall include as much information as can be determined by the system as to what caused the error, along with a suggested strategy for correcting that error.

6. Character by character syntax checking shall be performed as code is entered. The problem of forward referencing is to by handled by immediately informing the user that he or she is using a currently undefined symbol.

7. Extensive run time checking shall be performed by the visual execution module. Any run time error than can be checked for, will be.

8. The editor/assembler will format the source code as it is being entered.

9. The total OBVIAS system will be user-configurable. This configuration will be permitted during operation, but a provision for a startup file default configuration shall be included.
10. All efforts shall be made to ensure that OBVIAS is friendly and easy to use.

3. CS 400 Laboratory

Having established the initial specifications for OBVIAS, the next phase in the development process was to review the existing environment in which it is to be developed: the CS 400 Laboratory.

Briefly, the laboratory consists of eight Corvus Concept Personal Workstation computers, a 45-megabyte Winchester hard disk drive with a built-in network file server, a network print server, an Okidata 92 Microline dot-matrix printer, a single 8" floppy disk drive, and the Bank, a 200-megabyte cartridge tape storage device. All of these devices are interconnected via Omninet, Corvus' local area network.

For OBVIAS, the most important units in the lab would be the Concept personal workstations. Although designed for network use, the Concepts are powerful computers in themselves, composed of a large display, base unit and detachable keyboard.

The display is the most obvious feature of the Concept, and perhaps one of the most innovative. Measuring 15 inches, the display is fully bit-mapped and occupies 55k of main memory. Not only can the display tilt and rotate on the base unit, but it is made to function in two modes: landscape (horizontal), and portrait (vertical). In landscape mode, the display features a resolution of 720 X 560 pixels, which allows for a character resolution of 120

characters by 56 lines using a standard 10 X 6 pixel character set. In portrait mode, 72 lines of 90 characters are displayable. The large display area of the Concept, more than three times that of most micro computers, would become a pivotal factor in the success of OBVIAS.

One of the nicest features of the bit-mapped display was the ease of defining and using alternate character sets. The resolution of a character set can range from 1 X 1 to 16 X 16.

The other half of the Concept's user interface is the detachable keyboard. Patterned after the IBM Selectric keyboard, the Concept keyboard features a 15-key numeric keypad, 10 user-programmable function keys along the top edge, 4 cursor control keys and 62 traditional keys. In addition, every key is "soft," and may be changed simply by loading an alternate keyboard character translation table.

The base unit of the Concept contains a power supply, cooling fan and a tray containing a Motorola MC68000 microprocessor, either 256 or 512k bytes of main memory, four Apple compatible card slots, a speaker, clock battery, two serial ports and an Omninet tap connection.

The MC68k processor runs at 8 MHZ and features 18 32-bit registers, a 24-line address bus, a 16-bit data bus and one of the most powerful and versatile instruction sets available for microcomputers. Operations are permitted on data objects of 8, 16 or 32 bits.

Concepts can have either 256 or 512 Kbytes of main memory, but all of the CS 400 machines have 512 Kbytes.

The Concepts communicate with the outside world via the Omninet connection and the two serial ports. Additionally, the card slots can be used for local hard or floppy disk drive operations, as well as custom hardware applications.

The local area network that ties the laboratory together, Omninet, is a detection and retry network running on a twisted pair at 1 MHZ. The actual data transfer rate, however, is estimated to be about 60 percent of the clock speed because of packet transmission overhead. From the point of view of the Concepts, Omninet functions as a file and print server. Each Concept is responsible for its own processing, and Omninet is used as the viaduct for transferring data to and from the hard disk, and to the print server (via pipes located on the hard disk). Up to 64 network devices may tap into Omninet, and various device types are supported, including Rádio Shack, IBM, and DEC computers.

To summarize, three important features of the Corvus Concept stand out: the large display, the MC68k processor and Omninet. These features are fully exploited by the Corvus Concept Operating System, CCOS.

CCOS is a function key/windowed operating system emphasizing user-friendliness. When operating at the

topmost system, or operating system dispatcher level, four windows are displayed: the status window, containing the user's id, time and date; the large user window, in which the system and applications will do most of their communication to the user; the command window, a thin box near the bottom of the screen in which the user will type commands to the system; and finally the function key window, in which labeled pseudo-function keys are displayed, corresponding to the 10 user keyboard function keys. To perform most functions, a user may simply press a function key corresponding to the desired task. Because of the labeling of the displayed pseudo-keys, the keyboard overlays used by many systems are unnecessary.

Up to 17 user windows may be defined in addition to the four system windows. Each window may have its own character set, allowing the simultaneous display of multiple character sets. Only one window may be "active" at any time, but a user or application program can quickly switch between windows. Additionally, the windows may overlap.

The file system of CCOS is basically two-level: volume and file. A volume logically corresponds to a the structure of floppy disks on most microcomputer systems. It contains a directory, and the actual files. Further, files may be one of two types: text or data. As implemented, a volume is actually a contiguous chunk of the

hard disk or other mass storage medium, or an entire floppy disk. File allocation is also contiguous. Each user may have access to several volumes. Typically, a user will have read-only access to one or two system volumes, and read-write access to one or more personal volumes. Volume size is not restricted, but no more than 77 files may appear in any one volume. Other than access permissions to volumes, there is little protection. Users must enter a password to log on to the system, but after that no passwords are used. If a user has read-write permissions to a volume, he may do anything he wants to any file contained within it. It is possible to disallow a user any access to a volume by using the no-access permission.

Just as CCOS exploits the large display and power of the MC68k, so do the system programs supplied with it. The word processor, EdWord, can display a large amount of information on the screen, as can the spreadsheet program, LogiCalc. Additionally, all of the Corvus supplied programs conform to the function key/user friendly environment supplied by CCOS.

Pascal, Fortran, C and MC68k assembly language are available to the programmer, but Pascal is easiest to use. This is true for several reasons, irrespective of arguments about the inherent fitness or unfitness of Pascal as a development language. First, CCOS itself was written in Pascal, thus, it alone has the easiest operating system

interface capabilities. Second, virtually all of the interface documentation use Pascal examples. Indeed, what source code exists for informational purposes is almost exclusively written in Pascal. Most importantly are the supplied system libraries. Although similar libraries exist for Fortran and C, the Pascal library is the most complete, and was used extensively during the implementation of OBVIAS.

The primary Pascal system library, CCLIB, is composed of 13 units, each containing one or more related functions or procedures. A brief description of those units follows:

1. CCDEFN : These are the global system definitions that are used by the other units. Included are often-used data types and constants.
2. CCHEXOUT : This unit contains routines to convert and display numbers in hexadecimal format.
3. CCLNGINT : This unit contains routines to perform operations using the longword (32-bit) data type.
4. CCCLKIO : Also known as the clock control unit, CCCLKIO provides routines to set, read and manipulate the system clock.
5. CCCRTIO : One of the most often used units, it provides CRT functions such as clearing the screen, reversing the video display and positioning the cursor.

6. CCDCPIO : This unit handles the manipulation of the serial ports.
7. CCDIRIO : The directory unit, this provides routines to read and write volume directories.
8. CCGRFIO : Also known as the graphics unit, CCGRFIO contains routines to perform basic graphic manipulations such as setting points and drawing lines.
9. CCLBLIO : This unit controls the function keys. It allows the keys to be defined, displayed and manipulated.
10. CCOMNIO : The Omninet unit, this contains routines to allow a program to directly send and receive messages via Omninet.
11. CCWNDIO : Provides the ability to create and select display windows.
12. TURTLEGRAPHICS : As the name implies, this unit supplies the programmer with basic turtle graphic capabilities.
13. MISCELLANEOUS : This unit contains routines to perform low-level bit manipulation, check whether a key has been pressed but not read, and retrieve selected system parameters.

Another system library, C2LIB, is also provided. A brief description of its three units follows:

1. CCDRVIO : This unit provides access to the network disk server, allowing a program to make direct requests of the hard disk.
2. CCPIPES : Because of the access/no access protection schemes, this unit is particularly important. It provides a pipe mechanism, similar to the UNIX pipe. It is primarily used for print spooling.
3. CCSEMA4 : This unit enables a program to set up and test network-wide semiphores to protect critical regions in programs which might be running on several network stations simultaneously.

In addition to the Pascal libraries, the Pascal programming environment on the Concept is comprised of a Pascal compiler, a code generator, an object module library utility, a linker and an extremely low-level debugging program. However, the debugger is so limited that is essentially useless to the Pascal programmer.

The Pascal compiler was developed by Silicon Valley Software, and is their implementation of the Pascal language as defined in "Pascal News" [J]. SVS Pascal supports independent compilation units, using the UCSD notation. Further, it allows a programmer to define segments, which will be used as overlays by the operating system at runtime. The rule for overlayed execution is that unless a segment is locked into memory, it remains in

memory only so long as either code within it is executing, or code from a procedure or function within it is executing. The maximum size of an overlay is 32k bytes. This limitation is imposed in order to take advantage of the MC68k 16-bit relative addressing mode.

The Corvus supplied MC68k assembly language programming environment, which OBVIAS will attempt to improve upon, consists of a MC68k assembler, also developed by SVS, the linker, the object code librarian and MACSBUG [L], a rom-based debugger.

In practice, MACSBUG is never used by students. In addition to requiring communication via a separate terminal, its operating instructions were deemed too complex and confusing to burden the students with. In the first year of operation, MACSBUG was run less than four times - largely to make sure it functioned.

In addition to the Corvus-supplied environment, several demonstration and machine models were developed by students working under Dr. Schweppe in conjunction with the OBVIAS system. The most impressive, AMODE, was programmed by Alice Forester, an undergraduate, and Eric Harkness a graduate student. It provided a two-part tutorial on the MC68k addressing modes. The first part features a "fun" cartoon-style setting in which a character named Terry tries to locate a friend and illustrates indirect addressing. Next, the scene changes to "Terry and the

Pirates," as Terry sails to pirate island to follow instructions on a treasure map. The second segment of the tutorial is done in a more serious tone. It features a function key based tutorial in which students can select the individual addressing mode they wish to study. In addition to displaying dynamic examples on a MC68000 processor mockup, it asks questions and responds to student replies.

Also available are two prototype machine models: ZERO, a generic zero-address machine model, and THREE, a generic three-address machine model. In both models, students may select a brief tutorial, or may "program the machine" using a simplified instruction set. After programming the models, the programs could be visually executed slow, fast or single stepped. ZERO was programmed by undergraduate Jerre Bowen, and THREE was programmed by Allison Mills, a graduate student. Both students are developing second versions of their programs.

The last program developed at KU being actively used in CS 400 is SCREEN DEMON, a script-driven display and graphics management program by Hal Preston.

In addition to the programs already in use by students in CS 400, several additional models and tutorials are planned. To complete the battery of machine models, one and two-address machine models are planned, as well as a memory management model, a symbol table insertion/lookup

model, a Fortran syntax checker, a Pascal syntax checker and a data flow machine model. These tutorials and models, together with OBVIAS, will comprise the total CS 400 programming/learning environment.

Although related to the problem at hand only in an environmental sense, the CS 400 lab also features ISYS, an integrated word processor, spreadsheet program and graphic system; PAINT, a mouse operated graphic drawing system; PWX, a screen printing program developed by the author; TERMINAL, a terminal program; DISASM, an object code module disassembly program, developed by KU student Dennis Conley; CORE, a Pascal object code library implementing the CORE graphics standard, by KU graduate student Jim Buzbee; NEWQIX, a program which generates an infinite number of continuously changing and hypnotically beautiful graphic patterns, done by the author; and a host of system and graphic demonstration programs and utilities.

For further information on the Corvus system, consult Corvus [A], [E] and [F]. Documentation on AMODE, THREE, ZERO, NEWQIX, CORE, DISASM, NEWQIX and SCREEN DEMON is available through the Department of Computer Science at the University of Kansas.

4. Final Specifications

The next step in the development process was to draft the final specifications by mapping the initial specifications onto the hardware/software environment outlined in the previous section. This mapping, and the subsequent implementation of OBVIAS, was not a classic top-down process, but evolved into a modified bottom-up to top-down to bottom-up process.

The initial bottom-up phase involved immediately considering the actual implementation of the display format. This was deemed necessary because of the extensive visual requirements. Before proceeding further, it was necessary to verify that those requirements could be met using the Concept display. In fact, a display mockup was coded at this point - well before the bulk of the system specifications were even considered.

In considering the visual requirements of the system, it was necessary to isolate the displayable objects and determine how much display space they would require. Then, a prototype screen format was developed. This was an extremely valuable process, which resulted not only in a clean, professional display, but some important added features not considered earlier. A summary of the displayable objects and their individual requirements follow:

1. Source code text window: This would be the screen area in which the user would type in his code. Additionally, it would be used during the execution phase, when the code lines would be highlighted immediately preceding their execution. This window must include enough space for code and comments, line numbers, the machine code produced during the assembly process, and the relative address in which the machine code would lie. Further, it was decided to attempt to make this window look as much like a standard assembly listing as possible. An absolute minimum of 80 characters per line was determined.
2. MC68000 Microprocessor Model : The heart of the execution module, this area must contain the values of the eight data registers, the eight address registers, the status register and the user stack.
    a. Data Registers : Because the values in the 32-bit data registers are used in different ways depending on the program, it was decided to allow them to be displayed individually in any of four base combinations: hexadecimal/decimal, hexadecimal/octal, hexadecimal/ASCII and binary. Clearly, if a standard character set were used, allowing the display in binary would pose a problem. A

binary display would require a line at least 36 characters long. By using a smaller character set for the binary display, the width requirements of this field could be reduced to an acceptable level. A 4 X 10 character set would allow readable 0's and 1's, and satisfy the width requirement of this field 24 standard characters wide. The other three combinations would easily fit into such a field.

b. Address Registers : The address registers would follow the same format as the data registers.

c. Status Register : For the vast majority of ASM68k programming, the relevant bits in the status register are the carry, sign, zero, extend and the overflow. They could be displayed on a single line.

d. Stack : Although the stack pointer, A7, would be displayed in the address registers region, it was decided that the stack should be dynamically displayed in the same manner in which it is logically thought of by programmers. Thus, it would have a variable space requirement. The more room available, the more stack elements that could be

displayed.

e. Memory : Memory, or rather a user-defined window into memory, would also have a variable space requirement. The more room available, the more user memory could be displayed.

3. Help Manual : Some portion of the screen must be available to display the long help messages, or pages from the manual, as well as the extended error messages. This region, however, is not in continual use. For this reason it was decided to have the source code text window double as the help display region. This could be achieved by swapping out the source code when necessary, and restoring it after the help or error message had been read.

4. Command Entry/System Message : Some area of the screen would have to function as the command entry region. Although the function keys would be used for most user commands, some commands might require numeric or alpha entry. It was decided to simply use the existing CCOS system command window for this function, as well as for displaying short system messages.

5. User Program I/O : Because the programs written by the users would often contain screen output, some region would have to be set aside for this. Using the same reasoning as doubling the source code and

help regions, it was decided to devote the entire screen to user program I/O. Of course, this would be accomplished by swapping out the OBVIAS display before a user program performs screen I/O, and restoring it afterwards. This would have the added benefit of enabling a user to see exactly what his or her program output would look like if the program were not running in the OBVIAS environment.

After experimenting with different display configurations, it was decided to use the Concept in the horizontal, or landscape configuration, with the user source code window on the left and the CPU display on the right. Additionally, there was enough room left over in the CPU display to add another feature: addressing mode display. Two small windows, one for the source and one for the destination addressing modes, would be used to display the addressing modes used by the code lines.

Figure 4.1, on the following page, is an actual screen printout of the final display format.

Figure 4.1. OBVIAS display format.

Next, the other half of the user interface had to be considered. As previously decided, OBVIAS would be a function key based system, and the design proceeded with this in mind. This decision strongly influenced the remainder of the design, as well as the implementation. Because the function key approach is a variation on a menu-driven system, it lends itself naturally to a tree-like command structure. Once this command, or function key, tree was developed, it became obvious that the module design would match the command tree almost exactly. In fact, not only did the command tree become a general blueprint for the implementation of OBVIAS at the module level, but once expanded, it became a visual version of the final OBVIAS specifications.

The development of the command tree marked a return to the classic top-down strategy. Logically, the two most important tasks OBVIAS had to perform were edit/assemble, and execute. Thus, the topmost level in the command tree would have to include function keys to enter edit/assemble and execute.

The edit/assemble function key level was broken down in a similar manner. The important functions were determined and assigned function keys of their own. Such a function key would either be a terminal if it did not require a function key level of its own, or non-terminal if it did. Before proceeding, it is necessary to note that at

this time it was decided to limit the text editing commands to the insertion and deletion of lines. This was done in an effort to reduce the growing complexity of the system. Edit/assemble, henceforth referred to as the edit level, was broken down into the following functions: insert line, delete line and "workpad," an EdWord term referring to file manipulation. In keeping with the requirement to make OBVIAS as compatible with existing Corvus software in spirit as well as form, EdWord's terminology was adopted. A workpad, or pad, refers to an internal workspace in which users would type in separate assembly units. The text of the code in a workpad would be saved to a file, and a text file could be loaded into a workpad. The insert and delete line functions were determined to be terminal in nature, but workpad required its own function key level.

Workpad broke down into the following functions: save file, load file, make pad, clear pad, name pad, pad parameters and view pad. The purpose of each of these functions will be detailed later.

In the execution unit, there needed to be provisions for executing a program, controlling its run mode, setting display modes, setting breakpoints, setting the execution entry point, and manually modifying the CPU display. Of these, it was determined that execution initiation and run mode control could be treated as terminals, whereas entry point, breakpoint and CPU modification required further

function key levels. At this point, the supported run and display modes were specified. A program should be able to run single step, with breakpoints, or at variable speeds. Further, it was decided to add another run mode - micro step. This mode would simulate the fetch/execute cycle of the processor. The running display mode would be either trace on or trace off. With trace off, the CPU display would not be highlighted after each instruction is executed. Further, the code lines would not be highlighted. The run and display modes were also determined to be a function of the individual pads, and could be changed locally.

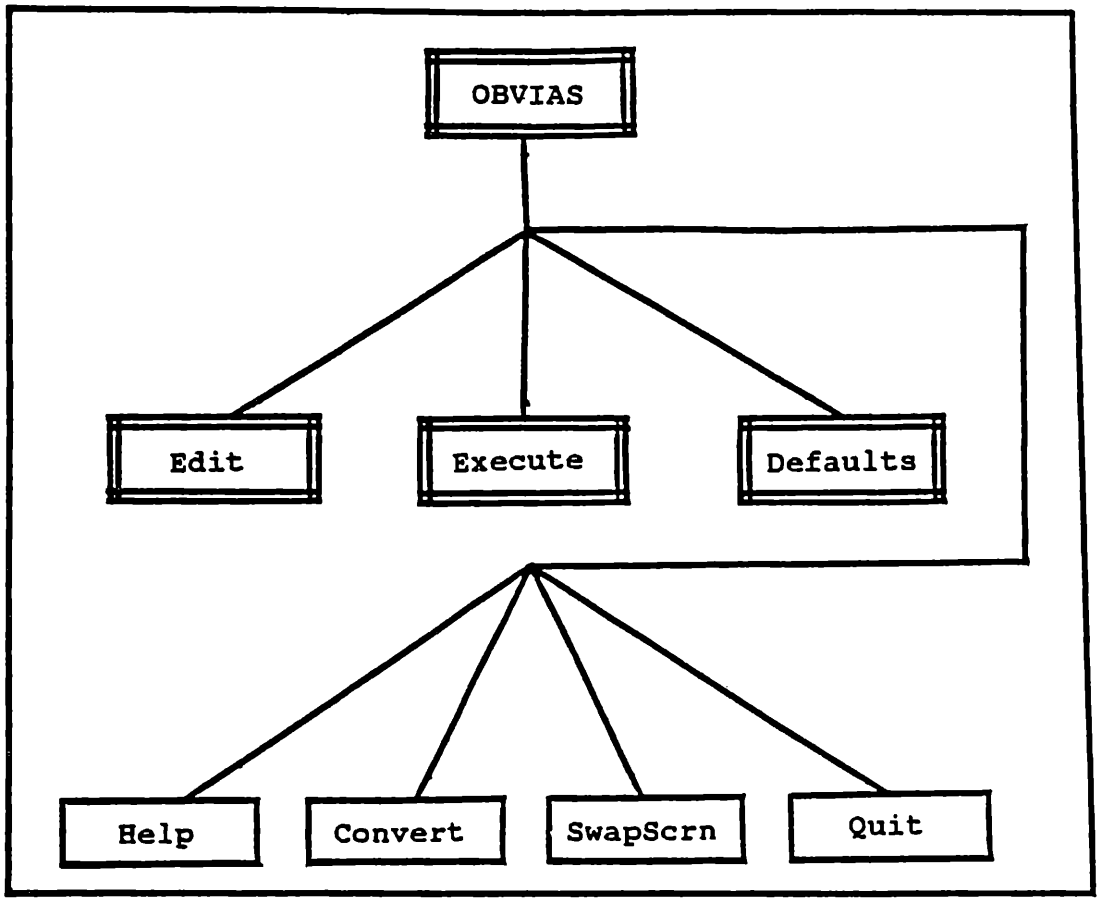
The entry point and breakpoint function key levels both needed a means of specifying the address in the program on which they were to operate on. Three modes were allowed: absolute address (something the user would probably not know), relative address (relative to the beginning of the current pad) and label. Further, it was decided that when a label selection was necessary, the function keys should be redisplayed, each one corresponding to a label in the user's program.

The CPU modification level would have to have provisions to change the value of the data registers, the address registers, that status register, values in memory locations and the register display modes. All of these functions were considered terminal.

Additionally, several modules were determined to have useful functions in multiple levels within the command tree. These were the help function, the default settings function, swap screen (temporarily exchange the OBVIAS and user program I/O screens), and convert - a generic number base conversion utility. Of these, only DEFAULTS was deemed to be encompassing enough to require a function key level of its own. These modules were inserted into the command tree wherever they were believed to be useful.

Minor modifications to the original command tree were performed throughout the implementation process, but there were no significant changes. For that reason, and for clarity, the final command tree will be presented in this section. In addition to a graphic representation of the level, non-terminal functions will be noted and terminal functions will be explained. This combination of a command tree and terminal descriptions will serve as the final set of OBVIAS specifications, as well as the implementation blueprint.

The command tree follows:



EDIT : Non-terminal, enter the edit/assemble module.

EXECUTE : Non-terminal, enter the execution module.

DEFAULTS : Non-terminal, enter the defaults module.

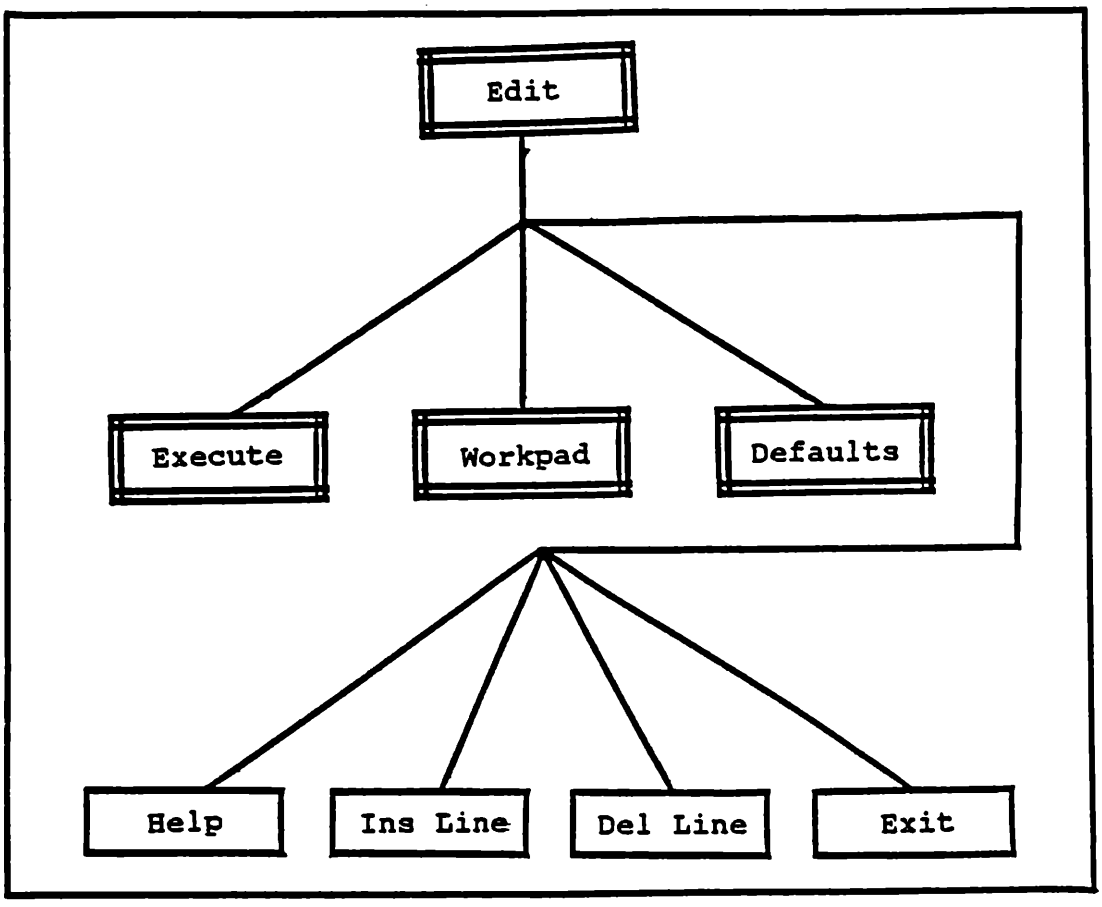
HELP : Terminal, access the system help function. This feature is to be provided in two forms: brief and expanded. Immediately upon entering HELP, the system will be in brief mode. Until the user exits by pressing the ESC key, any function key press will cause a short explanation of the key's function to appear in the command window. If a user then types "?", the user source code window will be swapped out

and a page from the users' manual will be displayed.

CONVERT : Terminal, enter the generic number base conversion utility. This utility will convert any 32-bit number in any base from 2 to 32 into any other base from 2 to 32.

SWAP SCREENS : Terminal, temporarily exchange the OBVIAS display and the user program I/O screen.

QUIT : Terminal, exit the OBVIAS environment, return CCOS.



EXECUTE : Non-terminal, enter the execute module.

WORKPAD : Non-terminal, enter the workpad module.

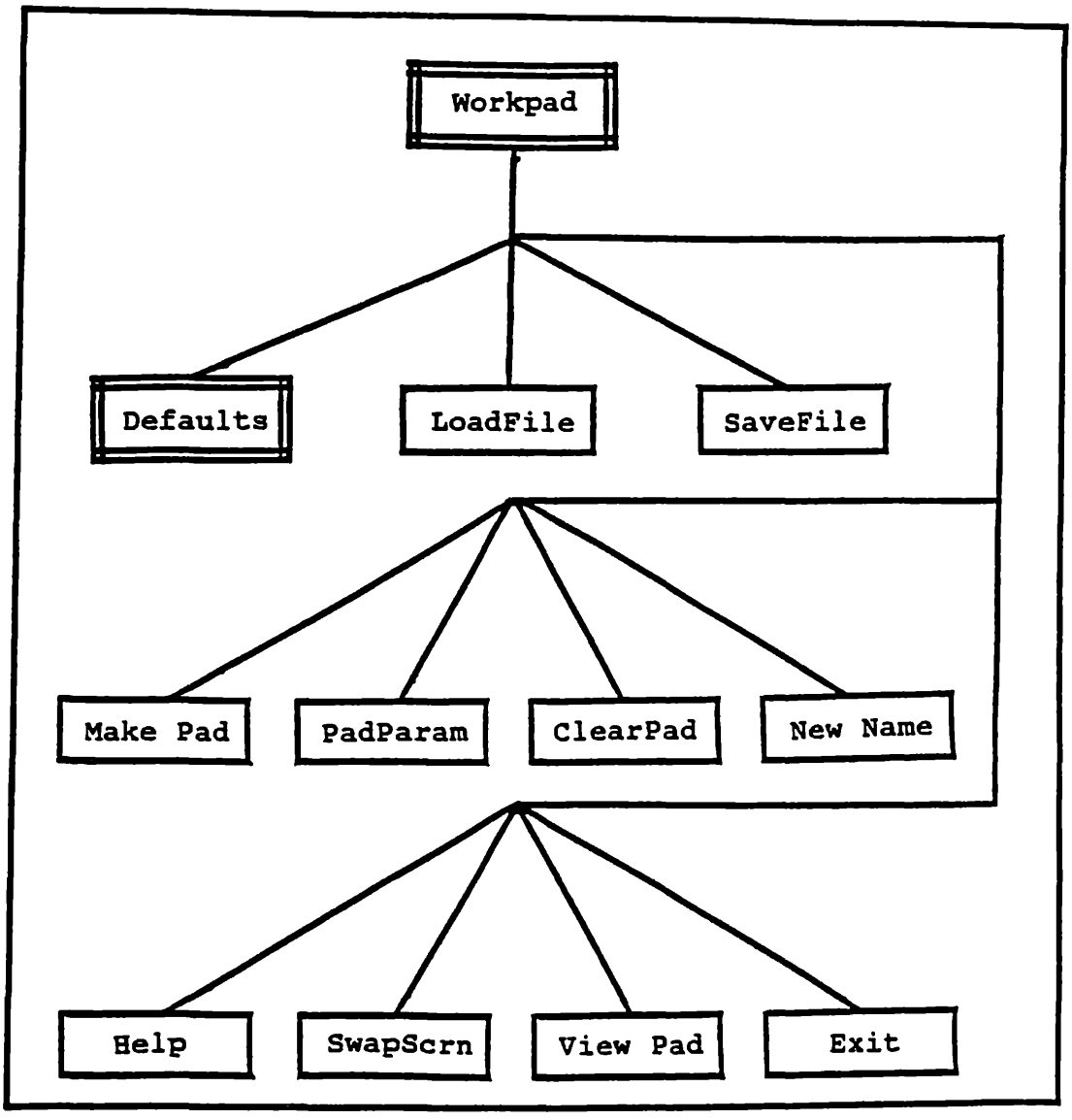
DEFAULTS : Non-terminal, enter the defaults module.

HELP : Terminal, go into help mode.

INSERT LINE : Terminal, insert source code lines. Although this is considered terminal, it actually does have two function key levels, but they are primarily for option selection and prompting.

DELETE LINE : Terminal, delete lines.

EXIT : Terminal, return to the calling function key level.



DEFAULTS : Non-terminal, enter the defaults module.

SWAP SCREENS : Terminal, swap OBVIAS and I/O screens.

HELP : Terminal, go into help mode.

LOAD FILE : Terminal, load a source code file into the current pad.

SAVE FILE : Terminal, save the source code in the current pad to a text file.

PAD PARAMETERS : Terminal, modify the run and display modes of the current pad.

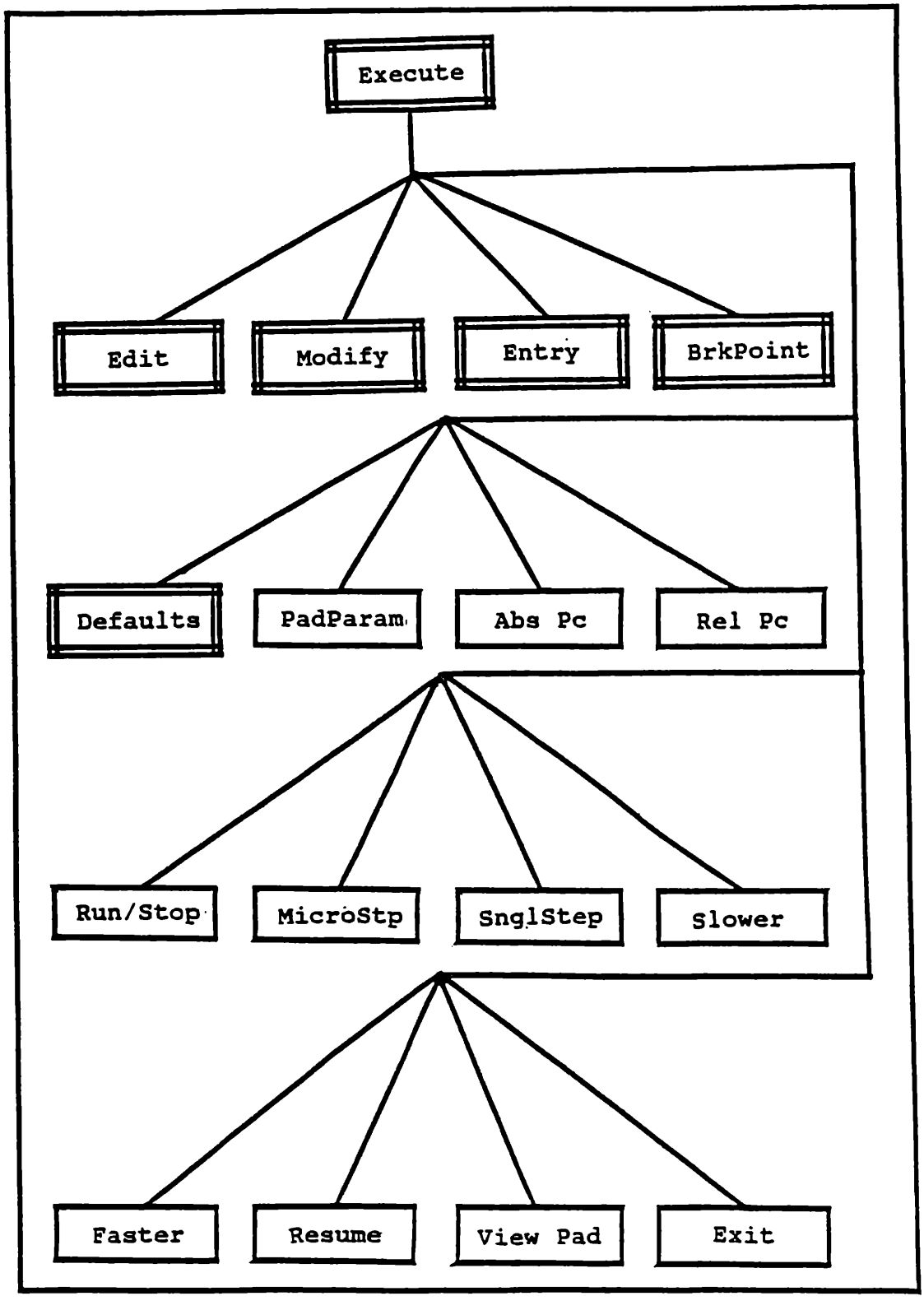
MAKE PAD : Terminal, create a new pad.

NEW NAME : Terminal, rename the current pad.

VIEW PAD : Terminal, make another pad "current."

CLEAR PAD : Terminal, clear all code from the current pad.

EXIT : Terminal, return to the EDIT function key level.


Execute
Edit
Modify
Entry
BrkPoint
Defaults
PadParam
Abs Pc
Rel Pc
Run/Stop
MicroStp
SnglStep
Slower
Faster
Resume
View Pad
Exit

EDIT : Non-terminal, enter the edit module.

MODIFY : Non-terminal, enter the CPU modification module.

ENTRY : Non-terminal, set the new execution entry point.

BREAKPOINTS : Non-terminal, set or reset execution breakpoints.

DEFAULTS : Non-terminal, enter the DEFAULTS module.

VIEW PAD : Terminal, make another pad current.

PAD PARAMETERS : Terminal, change the current pad's run and display modes.

ABSOLUTE PC : Terminal, display the absolute value of the program counter.

RELATIVE PC : Terminal, display the value of the program counter relative to the beginning of the current pad.

RUN/STOP : Terminal, begin or end program execution. Although this is considered a terminal, when a program is executing a subset of the applicable execute function keys will be displayed.

SINGLE STEP : Terminal, begin execution in single step mode, or, if running, change mode to single step.

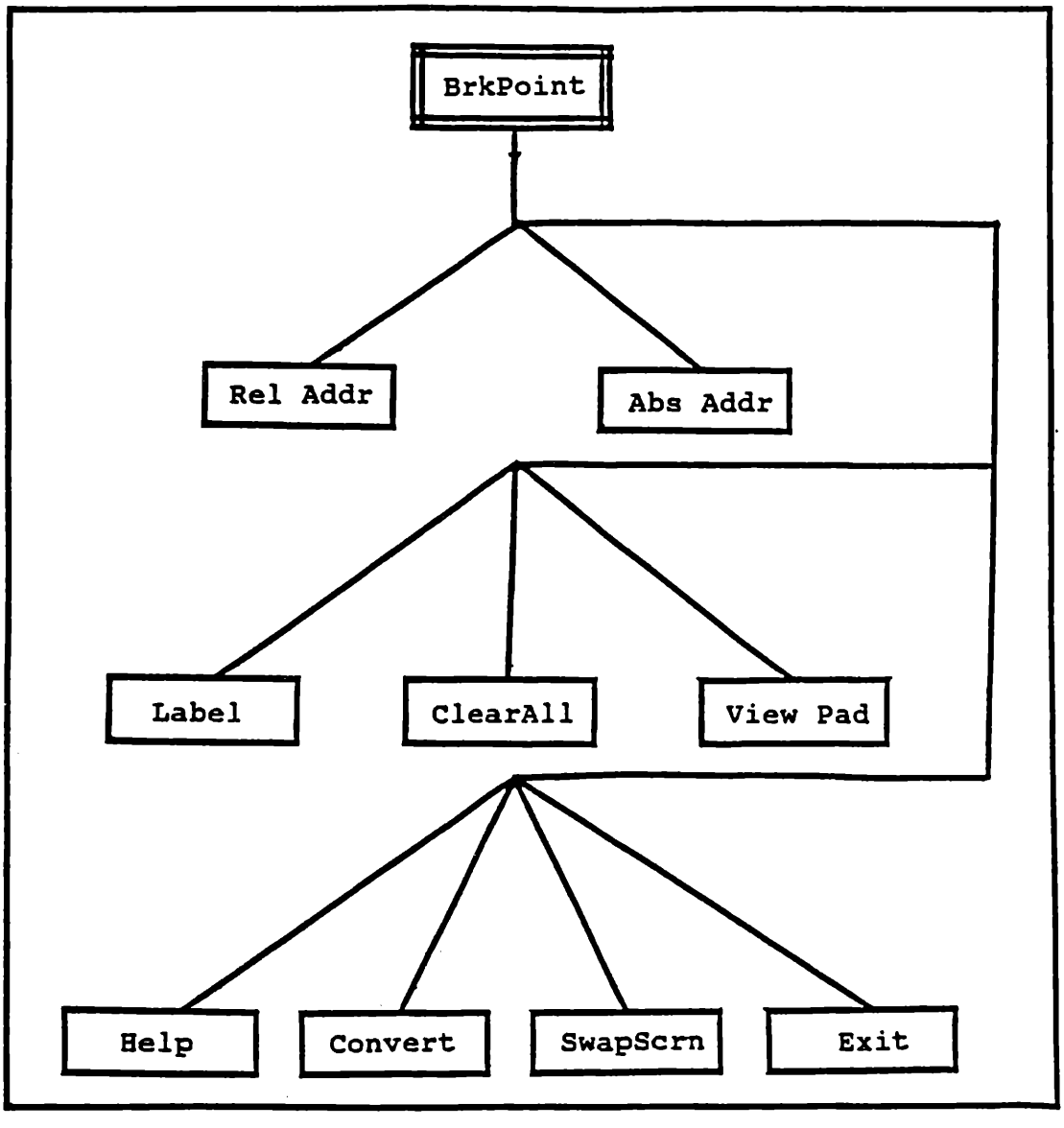
MICRO STEP : Terminal, begin execution in micro step mode, or, if running, change mode to micro step.

SLOWER : Terminal, slow execution.

FASTER : Terminal, speed up execution.

RESUME : Terminal, resume previous run mode following execution breakpoint.

EXIT : Terminal, return to the calling module.



CLEAR ALL : Terminal, clear all breakpoints.

LABEL : Terminal, set a breakpoint to correspond to a label in the program.

VIEW PAD : Terminal, make another pad current.

ABSOLUTE ADDRESS : Terminal, set a breakpoint to an absolute address.

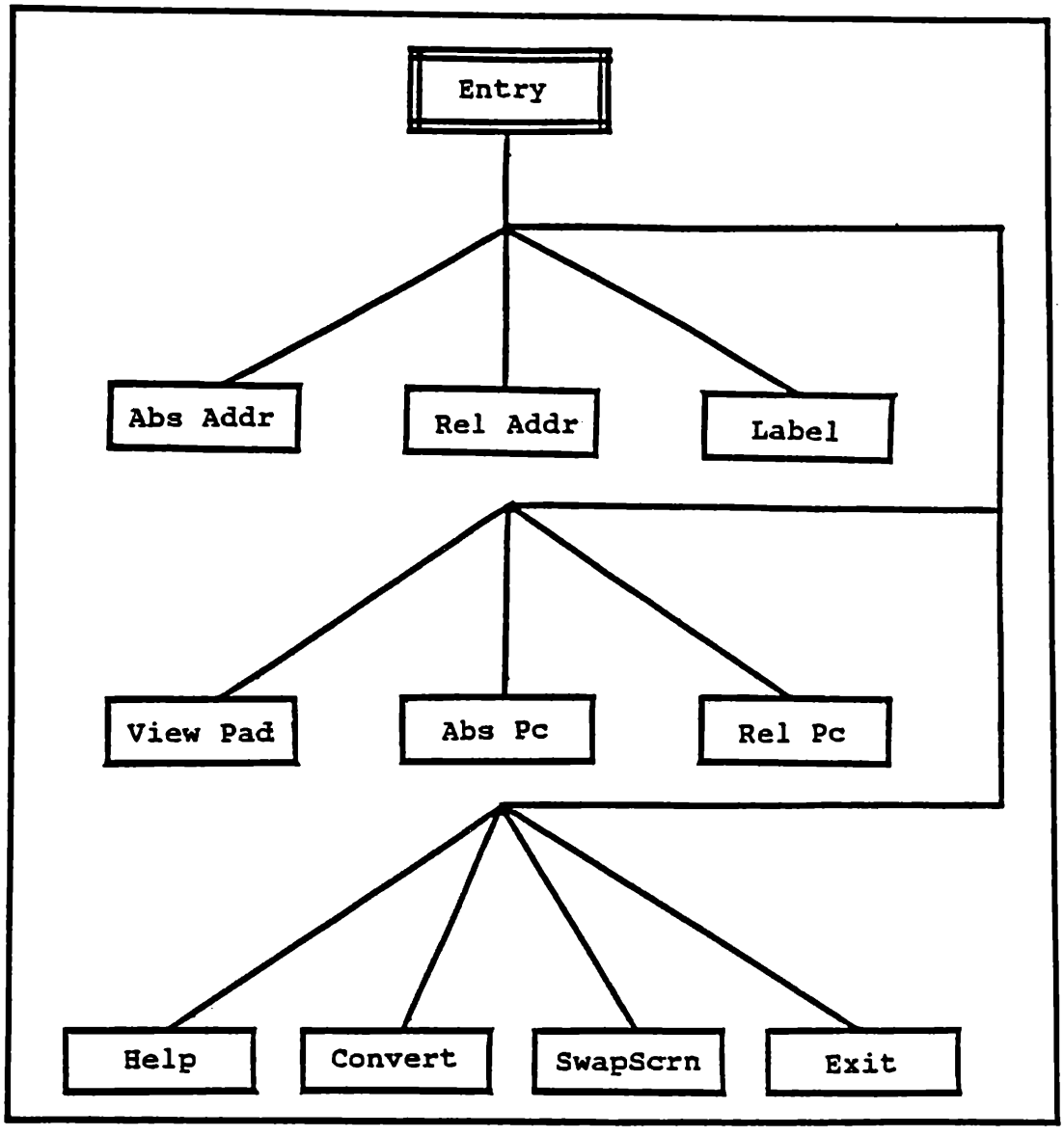
RELATIVE ADDRESS : Terminal, set a breakpoint to an address relative to the start of the current pad.

HELP : Terminal, enter help mode.

CONVERT : Terminal, enter the conversion utility.

SWAP SCREENS : Terminal, swap the OBVIAS and I/O screens.

EXIT : Terminal, return to the EXECUTE module.



ABSOLUTE ADDRESS : Terminal, set entry point to absolute address.

RELATIVE ADDRESS : Terminal, set entry point to relative address.

LABEL : Terminal, set entry point to label in program.

VIEW PAD : Terminal, make another pad current.

RELATIVE PC : Terminal, display the current value of the program counter relative to the start of the current pad.

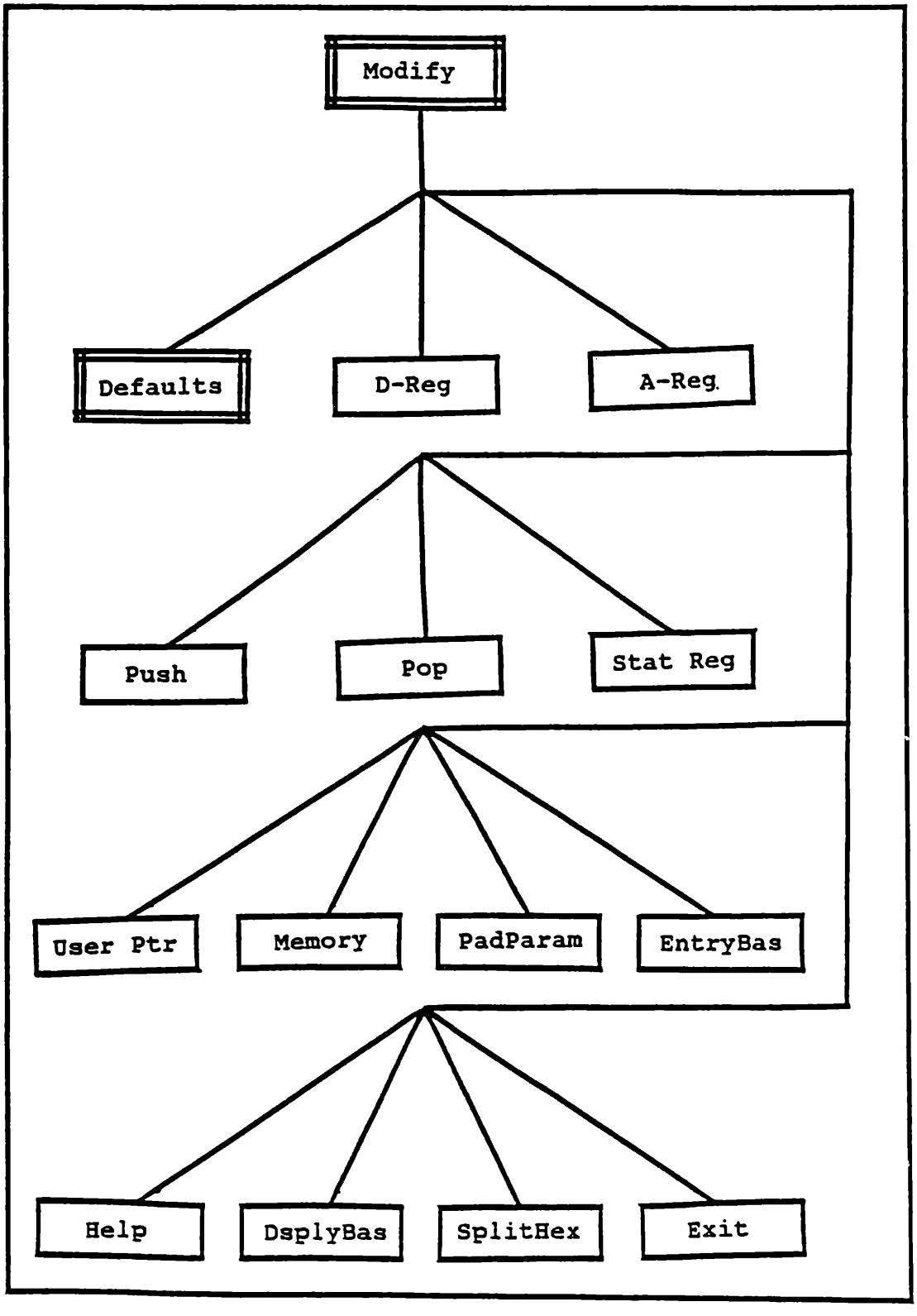
ABSOLUTE PC : Terminal, display the absolute value of the program counter.

HELP : Terminal, enter help mode.

CONVERT : Terminal, enter the CONVERT utility.

SWAP SCREENS : Terminal, swap the OBVIAS and user I/O screens.

EXIT : Terminal, exit to EXECUTE level.


Modify
Defaults
D-Reg
A-Reg.
Push
Pop
Stat Reg
User Ptr
Memory
PadParam
EntryBas
Help
DsplyBas
SplitHex
Exit

DEFAULTS : Non-terminal, enter the DEFAULTS unit.

ADDRESS REGISTER : Terminal, change the value of an address register.

DATA REGISTER : Terminal, change the value of a data register.

PUSH : Terminal, push a value onto the user stack.

POP : Terminal, pop the top value from the user stack.

MEMORY : Terminal, change the value of a memory cell.

STATUS REGISTER : Terminal, change the value of the status register.

PAD PARAMETERS : Terminal, change the run and display modes of the current pad.

SPLIT HEX : Terminal, toggle the display mode of the first word of machine code between normal hexadecimal and split hexadecimal[1]/octal[4] modes. In the split mode, the first four bits of the opcode are displayed as a single hexadecimal digit, and the last 12 bits are displayed as four octal digits. This enables a clearer picture of the structure of the actual machine code.

ENTRY BASE : Terminal, change the default entry base to any of decimal, hexadecimal, octal or binary. Note that when using numeric entry, the default entry base can always be locally overridden.

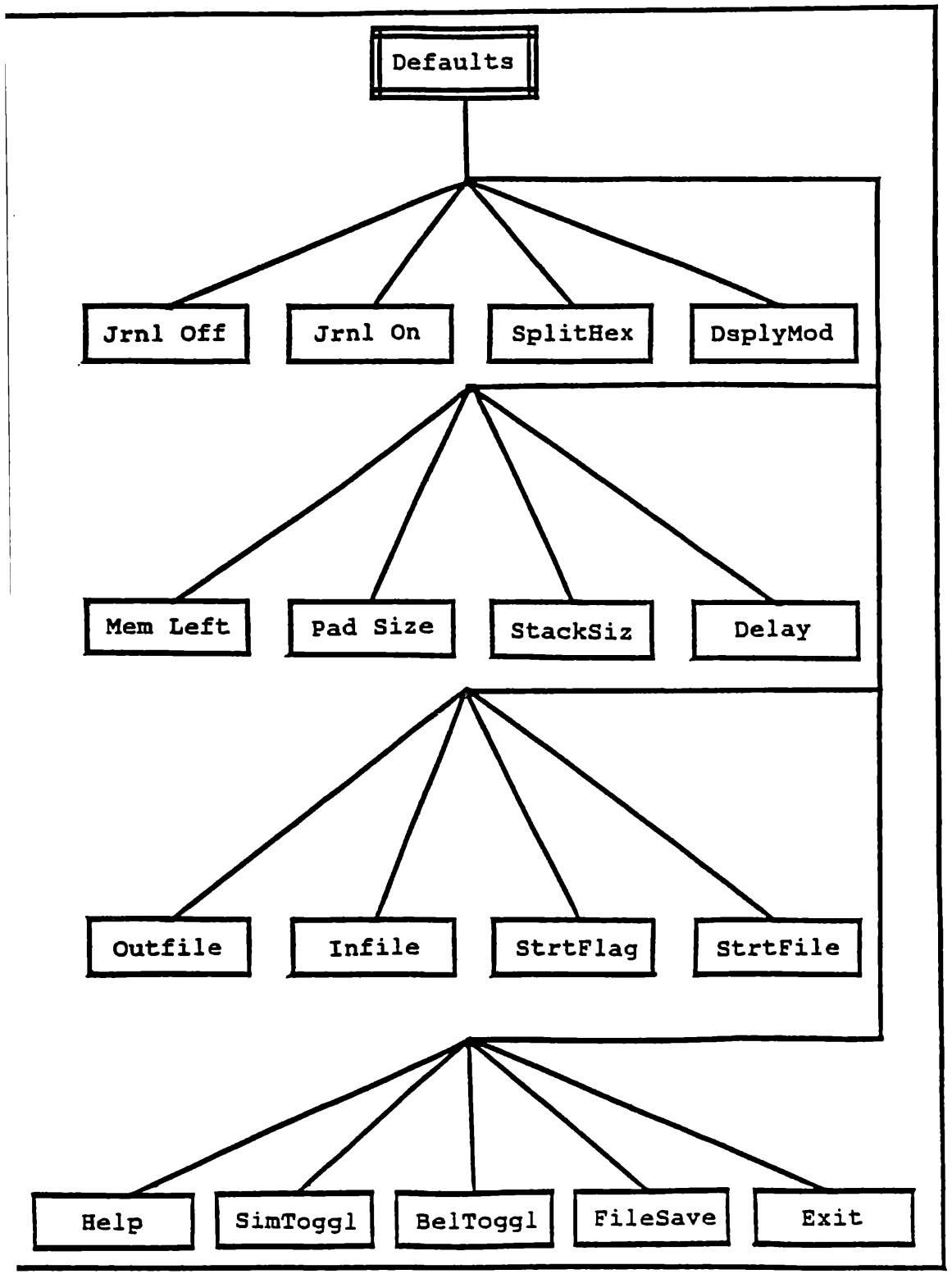
USER POINTER : Terminal, change the value of the user's pointer into memory to one of the following - absolute address, relative address, program label, or

dynamically linked to one of the address registers.

CONVERT : Terminal, enter the CONVERT utility.

SWAP SCREENS Terminal, swap the OBVIAS and user I/O screens.

EXIT : Terminal, exit to the EXECUTE module.


Defaults
Jrnl Off
Jrnl On
SplitHex
DsplyMod
Mem Left
Pad Size
StackSiz
Delay
Outfile
Infile
StrtFlag
StrtFile
Help
SimToggl
BelToggl
FileSave
Exit

MEMORY LEFT : Terminal, display the remaining machine memory in bytes and approximate code lines.

JOURNALIZATION ON : Terminal, begin journalization mode, sending all program output to a user-selectable file or device.

JOURNALIZATION OFF : Terminal, end journalization mode.

SIMTOGGLE : Terminal, set the default value of SIMTOGGLE.

DISPLAY MODE : Terminal, set the default value of the execution display mode.

SPLIT HEX : Terminal, set the default value of the split hexadecimal/octal machine code display mode.

BELL TOGGLE : Terminal, set the default flag controlling whether the bell will beep when an error is generated.

INFILE : Terminal, set the default file name from which user program file input will be read.

OUTFILE : Terminal, set the default file name to which user program file output will be sent.

HELP : Terminal, enter help mode.

PAD SIZE : Terminal, set the default pad size in Kbytes.

STACK SIZE : Terminal, set the default user stack size in Kbytes.

START FLAG : Terminal, set the flag that tells the system whether to load a source file immediately upon startup.

START FILE : Terminal, set the file name for the source code file that is to be loaded immediately upon startup if START FLAG is active.

DELAY : Terminal, set the default and current value for the variable execution delay variable.

FILESAVE : Terminal, save all default values, as well as the current state of the CPU to a default file that will be loaded upon program initialization. This enables a user to configure the system to his or her needs.

EXIT : Terminal, return to the calling module.

5. Implementation

The actual implementation phase of OBVIAS began in January, 1984. Some seven months and 14,000 lines of code later, the system was operational. Overall, the process went smoothly and quickly.

Since the emergence of Pascal and other block structured languages in the last two decades, the top-down programming methodology has risen to a place of dominance in the computing community. OBVIAS, with its tree-like architecture, seemed a natural for top-down implementation. This, however, was not to be.

The implementation of OBVIAS strictly followed the top-down canons in the early stages, but quickly transformed into a modified top-down, bottom-up development. The reason for this is simple: OBVIAS was required to perform tasks outside the scope of the design language, Pascal.

Top-down implementation requires that complex tasks be repeatedly divided into smaller, more intellectually manageable tasks. This division is to continue until the programmer is left with small, simple routines. Basically, it is the use of abstraction techniques which delay low-level decisions until the final stages of the implementation process. OBVIAS began in such a manner. The function key command tree detailed in the previous section was coded into the system's outer controlling

shell. Although none of the terminal functions were operational at first, the non-terminals permitted "movement" up and down the command tree. The idea was that the terminals could then be taken on one at a time, and handled in a similar manner.

At this point, the top-down method was unconsciously abandoned. OBVIAS required immediate resolution of numerous low-level implementation problems - fundamental decisions that could not be delayed. OBVIAS needed to directly access processor registers, recover from system run time error traps, directly manipulate the flow of execution, access specific memory locations and simulate concurrent processing. These tasks are not normally within the Pascal domain, and the structures of the higher level routines were intimately tied to the low-level methods of implementing them. Thus, the implementation process oscillated between top-down and bottom-up. The design of a terminal function would generally proceed top-down for a while, and then go bottom-up until the two designs met in the middle.

This was not a disadvantage at all. It worked quite well.

With one notable exception, the Pascal programming environment provided by Corvus worked well. The same large screen that would make OBVIAS so useful was of great help during the implementation of OBVIAS. Further, the unit

packaging and ease of implementing overlays was especially helpful. The one disadvantage was the SVS Pascal standard function library. Deficiencies in it forced the addition of more than 2,000 lines of code to OBVIAS - lines which never should have had to be coded. The problems stemmed largely from SVS' decision not to implement DISPOSE. Once dynamic memory was allocated, it could never be returned to the heap. Because of OBVIAS' large dynamic memory requirements, a complete heap management system had to be added. Further, the standard procedure READ had a tendency to cause fatal program aborts if an illegal character were inadvertently entered. Clearly, this was not consistent with the intended user-friendly environment.

The first problem of any consequence in the implementation phase was developing methods to "trick" the Pascal environment into performing the necessary low-level tasks.

Most of the low-level problems were directly related to Pascal itself, but one dealt with the operating system's window control mechanism. CCOS allows only one character set to be defined per window. OBVIAS, however, would have to display mixed character sets in various portions of the screen, and in the exact same character positions from time to time. Three methods were suggested to perform this.

CCOS determines the character set to use by following a pointer in a window control record. It seemed likely

that if OBVIAS changed this pointer directly whenever a new character set were needed, the problem would be solved. This method, however, did not work. There are several parameters in the window control record relating to character sets, and the exact purpose of them was not recorded in the available documentation.

The next method also involved directly altering the window control record. It was suggested that a single character set be created which included all of the characters which might be needed within a particular window. Those characters that were smaller than 10 X 6 would be "padded" with blank pixels on the top and left. When a normal sized (10 X 6) character was to be displayed, it would be written to the window in the usual manner. However, when a string of smaller characters was to be displayed, it would be written from right to left, and in between the displaying of each character, the cursor position variables in the window control record would be altered the desired number of pixels to the right. In other words, each smaller character would overwrite the left side padding of the previous character. This method did work, but it was not used in OBVIAS.

The method that was used in OBVIAS was really the simplest. CCOS required that only one character set be defined per window, but it had no restriction against more than one window covering the same portion of display space.

Thus, if three character sets were required in the same portion of the display, three windows would be defined to cover that space. Using this method, OBVIAS had only to select the appropriate window before displaying a character from the new character set.

There are several reasons that this method was chosen. First, it was the easiest to implement - not a trivial reason given the overall complexity of the system. More importantly it avoided direct manipulation of the window control record. The window control records are not intended to be modified by user programs. If Pascal had such a facility, the window control records would have likely been hidden in a data encapsulation package. There is no guarantee that a future version of CCOS might not handle window control records differently, therefore potentially causing problems to any program accessing them directly.

OBVIAS could avoid direct access of the window control records, but it could not avoid direct access of memory. In the execution unit, OBVIAS would have to support a running assembly language program. Not only would it have to place that program in a specific location, but it would have to be able to check for any memory cells that the program may have altered. In the SVS Pascal environment, there were two ways to accomplish this.

The system linker is capable of tying together object modules created by any of the supported compilers and the ASM68k assembler, provided the programmer supplies the correct calling sequence linkage. Thus, it would have been a simple matter to code two assembly language routines - one to store a value into memory given the address and data size, and another to return a value from memory, also given the address and data size. However, this method was not used. Instead, a Pascal variant record type was created to achieve the desired results. Its single variant field could be a long integer (32-bit), a pointer to a byte, a pointer to a word or a pointer to a longword. In short, it was a semantic trick to bypass the compiler's type checking mechanism.

The decision to use the "kludgey" variant record scheme was somewhat shaky, but there were valid reasons. First, one of the early implementation decisions was to keep as much of the code in Pascal as was possible. It is very easy to make mistakes in the Pascal to assembly calling sequence interface. Second, the MC68k microprocessor will not permit 16 or 32-bit memory access beginning on an odd address. It was easier to include the error handling and recovery in the Pascal versions of the fetch and store routines than it would have been in assembly language versions.

The disadvantages to the method chosen should not be overlooked. The SVS compiler does not support code optimization, but if a future version did, OBVIAS might no longer function. For example, the following code sequence retrieves a byte of data from memory location 1000.

```
VARREC.N := 1000;      (* Longword portion of VARREC
                          set to address *)
I := VARREC.B;         (* Use as pointer to byte *)
```

A code optimizer might not recognize that VARREC.N is really an alias of VARREC.B, and delete the assignment. In any event, a potential for problems exists. These problems would have been avoided had assembly language language routines been used.

Assembly language was used to solve the next problem: direct manipulation of the processor's internal registers. The register contents of the Pascal environment and the register contents of the user's program would have to be exchanged before and after each line of code in the user's program was executed. Further, after execution, OBVIAS would need to compare the new register values with the old ones to determine if the CPU display needed to be updated. This was accomplished by using several assembly language routines. The register swapping would be handled entirely in assembly language. Another assembly language routine would return a pointer to the register save area. This

pointer would be assigned to a pointer variable using a record template that matched the format of the register save area. OBVIAS could then manipulate that area using Pascal.

The most complex task of the execution unit was to force the processor to actually execute each code line in the user's assembly language program. This was accomplished by using special trap handlers and the MC68k trace trap bit. Motorola included a special single step trace feature in the MC68k. When the trace bit in the program status word is set, the processor will execute the next instruction and then perform a special system trap. Upon execution of the system trace trap, the processor saves the program counter and the current status word, goes into supervisor mode and jumps to the trap handler whose address is stored in the trace trap vector.

The solution to OBVIAS' problem would be to load the processor with the user's register values, and then simultaneously insert the address of the instruction to be executed into the program counter and set the trace bit. After the instruction is executed, the trace trap would occur. OBVIAS would have to replace the system's trace trap handler with one of its own - one that would save the user program's registers and status, restore the Pascal environment's registers and status, turn off the trace function and return to the Pascal environment.

The two primary difficulties with this solution were how to simultaneously change the program counter and the trace bit, and how to return to the Pascal environment. Both were solved by using a secondary trap mechanism. As stated earlier, the initiation of a trap causes the processor to save, at minimum, the current program counter and status word on the stack. The return from trap instruction, RTT, simultaneously restores the previous values of the PC and the status word by popping them from the stack.

Following is the algorithm used to leave the Pascal environment, execute one instruction in the user's program, and return to the Pascal environment:

0. (From Pascal environment) : Make a copy of the register values of the user's CPU model and save them for later comparison.

1. (From Pascal environment) : Call assembly language routine TSETUP.

1.1 (From TSETUP) : Store the address of assembly language trace trap handler, THANDLE, in the system trace trap vector.

1.2 (From TSETUP) : Store the address of assembly language user trap #0 handler, TSETUP, in the system trap #0 vector.

1.3 (From TSETUP) : Execute return from subroutine instruction.

2. (From Pascal environment) : Call assembly language routine EXEC1LINE. Note: the subroutine jump mechanism will save, on the stack, the address of the next instruction to be executed in the Pascal environment.

2.1 (From EXEC1LINE) : Execute TRAP #0 instruction. Note: this will cause the program counter and status word to be pushed onto the stack, and the new program counter to be taken from the TRAP #0 vector.

2.1.1 (From TSTART) : Save the current values of all CPU registers. Note: these values are the Pascal environment's registers, and have not been changed since the call to EXEC1LINE in step 2.

2.1.2 (From TSTART) : Remove the saved status word from the top of the stack and save it. Note: this is the Pascal environment's status word, also unchanged since the call to EXEC1LINE in step 2.

2.1.3 (From TSTART) : Remove EXEC1LINE's return address from the top of the stack and discard it. This return address will not be used.

2.1.4 (From TSTART) : Remove the Pascal environment's return address from the top of the stack and save it. This return address will be used later

to return to the Pascal environment.

2.1.5 (From TSTART) : Push the address of the instruction in the user's program that is to be executed.

2.1.6 (From TSTART) : Push the user program's status word onto the stack. Note: steps 2.1.5 and 2.1.6 have effectively replaced EXEC1LINE's return address and status word with those from the user program.

2.1.7 (From TSTART) : Set bit 15, the trace enable bit, of the user status word that is currently located on the top of the stack. Note: this does not affect the current status word.

2.1.8 (From TSTART) : Load all CPU registers with the user's program CPU values.

2.1.9 (From TSTART) : Execute return from trap instruction. Note: this will cause the processor to pop the status word and return address from the top of the stack. Execution will then commence, in trace mode, at the target instruction in the user's program.

3. (From User's Program) : Execute one instruction.

3.1 (From User's Program) : After instruction is executed, a trace trap will occur, pushing return address and status word, and continuing execution at the address stored in the system

trace trap vector.

3.1.1 (From THANDLE) : Save all CPU register values. Note: these are the new register values of the user's program.

3.1.2 (From THANDLE) : Pop the user's status word from the stack and save it.

3.1.3 (From THANDLE) : Reset the trace enable bit in the user's status word that has just been saved.

3.1.4 (From THANDLE) : Pop the return address from the stack and save it. This will be the new value of the user's program counter.

3.1.5 (From THANDLE) : Push the Pascal environment's return address onto the stack. Note: this was the return address created in step 2 and saved in step 2.1.4.

3.1.6 (From THANDLE) : Push the Pascal environment's status word. Note: this was the status word saved in step 2.1.2.

3.1.7 (From THANDLE) : Restore the Pascal environment's register values. Note: these are the register values saved in step 2.1.1.

3.1.8 (From THANDLE) : Execute return from trap instruction.

4. (From Pascal environment) : Compare the current values of the user program's CPU with the values

saved in step 0, and update the CPU display if necessary.

The actual code used in OBVIAS was slightly more complex than the above example because of the requirement that all run-time errors be handled. If an error trap, such as an invalid address, occurs during execution in trace mode, it will take precedence. To solve this problem, all of the system error trap handlers also had to be replaced with custom OBVIAS versions. The error trap handlers would function identically to THANDLE, with the exception that they would set a run-time error variable that would be visible within the Pascal environment. Additionally, whatever diagnostic information that the trap provided would also be passed to the Pascal environment.

The last major problem involved the implementation of the edit/assemble unit: how would OBVIAS perform simultaneous character by character syntax checking and incremental assembly.

In reviewing the final specifications, the edit/assembly unit would have to:

1. Completely syntax check the user's program on a character by character basis during entry. Any invalid character will be discarded. To handle the problem of forward referencing, the currently undefined symbol will be immediately highlighted to

inform the user that it must be defined later. When it is defined, the highlighting will be removed.

2. If desired, and if possible, OBVIAS will execute a code line immediately following entry. This immediate execution is not possible if the code line uses an undefined symbol, or if it is a branch, jump or pseudo operation.
3. At all times during entry, OBVIAS will present a prompt line showing the user what is expected. At the beginning of the line the user would be told that a label, space, comment or carriage return was possible. Further, when the user is typing a label or comment, the syntax of labels and comments will be displayed. When the user is typing an opcode or operand, all possible choices will be displayed, and updated after each new character is accepted. For example, if the user had typed "AD" in the opcode field, ADD, ADD.B, ADD.L, ADD.W, ADDA, ADDA.L, and ADDA.W would be possible.
4. At any time during entry, the user can request help and OBVIAS will decide what information the user most likely needs. For example, if the user has typed "ADD.B" in the opcode field and requests help, OBVIAS should retrieve the full description of the ADD.B instruction from the programmer's

manual and display it. Likewise, if the user is typing a label and requests help, a full description of labels, complete with examples, will be displayed. If help is requested in the opcode or operand fields and more than one choice is available, OBVIAS will present, in alphabetical order, the manual pages covering all choices.

5. OBVIAS will format source code as it is entered.

Considering only the syntax checking requirement, the operand fields would pose the only real problem. During label entry, OBVIAS could simultaneously search the symbol table, and would know immediately when a label had not previously been defined. Once comment entry begins, any combination of keystrokes is syntactically correct, so it posed no problem. The checking of opcodes is also relatively simple. The opcodes would be stored in a sorted table, which could be searched after each new character was entered.

The problem with the operand field stemmed from the wide variety of permissible operands. Most operation codes permit six to eight different operand types in both source and destination fields. Further, many of those addressing modes can include constant or relative expressions. Additionally, the allowable addressing modes for the destination field might change depending on what addressing

mode was used in the source field.

Clearly, ordinary programming methods would be insufficient for such a complex task.

Two programming language syntax checkers have previously been implemented at the University of Kansas by graduate students working under Dr. Schweppe at the University of Kansas.

Designed by Dr. Schweppe and programmed by John C. Pinc, a Fortran statement by statement interactive syntax checking/prompting system [C] became functional in 1973. This system was developed in an ad hoc manner on a very limited Datapoint 2200, Version 1, with a serial arithmetic/logic unit and 8 Kbytes of shift register memory. The system would provide users with a dynamic prompt which displayed a top-down abbreviated syntax description during code entry. As each character was typed, the prompt would be updated. Further, if the current input completely specifies the target statement, the remainder of the statement will be supplied by the system.

An almost complete ANSI Fortran 66 system based on this work was developed in 1976 by Mary Owens Cheng [M]. This system used transition matrix techniques and the whole system occupied less than 8 Kbytes of memory, even though it was developed on a 16 Kbyte machine.

The transition matrix method would not be appropriate for OBVIAS not only because the large number of addressing modes and expressions would force the creation of extremely large tables (or perhaps a large number of smaller tables), but because it was suspected that the debugging process for such tables would be oppressively long and arduous. This concern was expressed by Roehl [K] in reference to the PDP-11 visual emulator. Although it supported only a limited subset of the language (70 mnemonics and limited operands), a transition matrix of more than 5,000 entries was required. OBVIAS would have to support more than 240 different operation codes and considerably more complex addressing modes. A transition matrix for OBVIAS would be enormous.

It was decided to use a production-driven system. In such a system, a small pseudo-machine would be implemented as an assembly language driver. This machine would be composed of a stack and a program counter, and would interpret productions stored as data statements. The productions would be simple, allowing a comparison of data on the stack with a string of tokens. If a match occurred, the matched string on the stack would be replaced by a string specified in the production, and a list of actions would be performed.

For example, the following is a sample production which will perform syntax-checking for the data register

direct addressing mode (D0,D1,D2...D7). Action GETCHAR takes the current input character and places it on the stack, action GOTO xxx causes the driver to continue interpretation at label xxx, action ERROR performs error recovery, and action ACCEPT accepts the string and returns. Interpretation will begin at label DDIR.

```
(LABEL) (ON STACK?) (REPLACEMENT)   (ACTION LIST)
=====================================================

DDIR:                                / GETCHAR
      "D"           =                / GOTO DDIR1
      (ANY)         =                / ERROR; GOTO DDIR

DDIR1:                               / GETCHAR;
      "0".."7"      =    (SAME)      / GOTO DDIR2
      (ANY)         =                / ERROR; GOTO DDIR1

DDIR2:                               / GETCHAR
      (EOL)         =                / ACCEPT
      (ANY)         =                / ERROR; GOTO DDIR2
```

In the above example, only a "D" will be allowed on the first keystroke. Any other character will be discarded. After a "D" is entered, any digit between "0" and "7" will be permitted. Any other character will be discarded. Finally, the production requires that an end of line character be entered before the addressing mode is accepted. The numeral denoting the data register used is left on the stack so that it can be used in the assembly process.

For most of the MC68k addressing modes, the stack in the production driver is not really necessary. The data and

address register direct, address register indirect, indirect with post-increment and pre-decrement and register list could have been checked using only the next available character. However, the stack becomes indispensable when constructing productions to handle expressions.

Having decided to use productions, the next step was to determine how many productions would have to be developed - and how complex they would be. Strictly following the above scheme, a set of productions would have to be developed for every combination of addressing modes permitted by the various operation codes. For example, to cover the source operand field of the ADD.B instruction, the production would have to permit one, and only one, of the following addressing modes:

1. Data register direct.
2. Address register indirect.
3. Address register indirect with pre-decrement.
4. Address register indirect with post-increment.
5. Address register indirect with displacement.
6. Address register indirect with index.
7. Absolute short address.
8. Absolute long address.
9. Program counter with displacement.
10. Program counter with index.
11. Immediate data.

Clearly, a production to handle this would have been complex. Further, it was determined that more than 30 combinations existed - all but two or three similarly complex. This complexity was not just an implementation concern, but it raised fears that the debugging process would be long and unstable.

Ideally, a tool could be developed to automatically generate the productions given the syntax of the individual addressing modes. Such a tool, however, was not available and the development of one was beyond the scope of this project.

The problem, then, was to reduce the complexity of the productions to the point that they were simple enough to confidently debug. This was accomplished by modifying the production drivers to simulate concurrency.

The new approach would be to generate one production for each single addressing mode. Each of these productions would be generally small and simple. Then, rather than using one production for each combination of addressing modes, a combination of productions would be used "concurrently."

Considering the problem abstractly, each addressing mode production would constitute a pseudo machine. At the beginning of the syntax checking, each applicable machine would be activated by sending it the first input character. Concurrently, each machine would operate on that character

and make the appropriate transition. If the transition resulted in an error, that machine would die. After the first input character had been operated on by all the machines, the machines that were still alive would be sent the second character and would again make the appropriate transition. This would continue until one of the machines announced that it had accepted the input string - i.e. a syntactically correct addressing mode had been entered. If all of the machines died on a particular input, that character would not be accepted and the machines would restart at their previous state.

This method not only had the benefit of simple productions, but it made it easy to determine which addressing modes were still permissible given the previous input. In order to construct the required prompt line telling the user what choices are still available, OBVIAS would have only to check to see which machines were still alive.

The only problem with the use of "concurrent" productions was the implementation of the pseudo-machine driver. The driver would have to take the current character, make the appropriate transitions, save its state within the current production, and then operate on another production. It would also have to recover the previous state in case a transition resulted in an error.

It was decided to develop one generic driver that would be appended to each of the productions. This driver would have to be coded in assembly language in order to save its state across subroutine calls. Additionally, a companion initialization routine would be developed for each driver/production combination. This initialization routine would be called before the first input character was received to reset the driver.

Besides doing syntax checking, OBVIAS would also have to perform incremental assembly during code entry. This was done in three steps.

First, after the operation code was correctly entered, a record would be retrieved containing information of what operand types were permissible and what assembly format the instruction used. Additionally, the record would contain a machine operation code template. This 16-bit number would be filled in with the correct value for the operation code, with the operand fields left blank.

Next, OBVIAS would use the operand type information to determine which addressing mode pseudo-machines to activate. Besides performing the syntax checking, some actions in the addressing mode productions would be devoted to saving pertinent information for the assembly phase. For example, the data register direct mode would save the number of the data register. After the operand was completed, this information would be passed back to the

edit/assemble unit.

Finally, the operand fields in the machine code template would be filled in using the information provided by the syntax-checking productions. This process would be repeated for for two-address instructions, and the display would be updated whenever possible.

If a code line could not be assembled because it used a forward-referenced symbol, it would be marked as unfinished, and linked into a chain associated with the undefined symbol. When that symbol was defined, the chain would be followed, and each affected code line would be reassembled at that point.

One final edit/assemble implementation decision should be discussed: how backspacing was handled. The productions were designed to make transitions in one direction only - forward. But to be truly interactive and useful, the system should be able to handle backspacing. Once again, the simplest method was chosen. A copy of all previously entered characters is maintained during entry. If a backspace is encountered, the last character entered is deleted from the copy, edit/assemble is initialized, and that string is fed back through the entire process. During string re-feeding, the display is not updated to avoid a delay. As implemented, the code runs fast enough that string re-feeding does not produce a noticeable delay.

The core of OBVIAS consists of the execute trap mechanism and the concurrent productions, but surprisingly, they were among the easiest features to implement once the solutions were discovered. More than 80 percent of OBVIAS' total code is devoted to producing a user-friendly environment. Despite the problems detailed in this section, the most difficult task from a coding aspect was generating the routines to make the system polished and professional. Many features not included in the final specifications were added during the coding process. The value of these features should not be overlooked despite their comparative lack of sophistication. The cumulative effect of the "polish" routines would determine whether OBVIAS would be a useful system or just an interesting side attraction. Among those features were:

1. Variable base numeric entry: Users are given the choice of selecting decimal, hexadecimal, binary or octal as the default numeric entry base. Further, they are given the ability to locally override the default base by preceding their entry with a special character.
2. Visual stack: OBVIAS' stack visually functions like a stack should: it grows up and down within its window. Further, it remembers the size of the data object pushed and displays the value accordingly.

3. Convert utility: A generic number base utility is provided on most function key levels. This utility will convert a number in any base from 2 to 32 into any other base from 2 to 32.
4. Pad size control: Users may control the size in Kbytes of the pads they create as well as the default pad that is created upon system initialization.
5. Stack size control: Users may control the size in Kbytes of the user stack that is created upon system initialization.
6. Label selection: When a user symbol selection is required, the symbols will appear on the function keys sorted alphabetically.
7. Pad renaming: A provision is included to rename user and system pads.
8. Address selection: Whenever an address selection is required, users are given the choice of specifying it as an absolute address, relative address or label.
9. User pointer: The user's window into memory pointer can be set to an absolute address, relative address, label or dynamically linked to any of the address registers.
10. Journalization: A journalization feature is included to permit the user to echo all program

output to a disk file, printer or other device.

11. Editing keys: While typing in source code, a user may delete the current field or restart the line from the beginning by using special function keys.
12. Bell toggle: A bell rings whenever a system error occurs, but it may be turned off by users who find it annoying.
13. Expert mode: Experienced users can disable the code entry prompting by invoking expert mode.
14. Memory left: The available system memory is displayed not only in bytes, but in approximate code lines.
15. Start file: A source code text file, perhaps containing library routines, may be automatically loaded upon initialization.
16. System configuration: The total OBVIAS system configuration may be saved to a personal or system-wide default file that will be loaded during system initialization.
17. File robustness: If any of the files needed for system execution are not available, a detailed error message will be displayed. Further, the system can still be operated provided the opcodes file exists.
18. Personal files: OBVIAS will always look for help and error message files first on the current

volume, and then on CCSYS. Because of this, each user on the network could have his own personal error or help files, as well as his own personal default settings file.

19. Run-time checking: Extensive run-time error detection and recovery is performed.
    a. Code corruption: If a user program corrupts its own code, the user is warned and the code is corrected.
    b. Stack over/underflow: The user's stack is checked for both underflow and overflow.
    c. Data size mistake: Because the stack remembers the data size of the objects placed on it, OBVIAS requires that they be taken off in the proper manner. For example, if a user pushed a long word, he cannot pop a word.
    d. Proper code execution: Only code lines in a user's program may be executed. Attempts to execute data, pseudo operations or system code are prohibited.
    e. Address violation: OBVIAS will recover from a user address violation and display a detailed error message.
    f. Bus time out: OBVIAS will recover from a bus time out and display a detailed error message.
    g. Additional run time error traps: OBVIAS will

detect and recover from privilege violation, illegal instruction, division by zero and reserved instruction traps.

To summarize, although the trickiest part of the implementation was developing solutions to the core of the system, the most difficult task was generating the thousands of lines of code to make the system friendly and easy to use.

6. Data Structures

In contrast to the cleanliness of the hierarchical OBVIAS module design, the initial attempt to define the system data structures resulted in an conglomeration of similarly named data objects. The reason is that OBVIAS required a large number of globally defined variables. The CPU model would have to be visible to virtually every module in the system, as would the user program structure, symbol tables and the system parameters.

The primary concern was not one of computational functionality, but of programmer understanding and system maintainability. The static global variables would be handled similarly by the compiler no matter how they were defined, but it was suspected that the large number of globals would cause considerable confusion during coding and maintenance.

The solution chosen was to cluster data objects with similar functions into large global records. In effect, this would modularize OBVIAS' data structure at the global level. This philosophy of data structure modularization was most easily applied to static objects, but was applied to the dynamic structures as well, generally through the use of header records on lists.

The packaging of OBVIAS data objects corresponds to a conceptual view of the system. The primary data objects in the system are the CPU model, the user program, and the

state of the system. These were broken down into the following packages:

1. CPU: Conceptually represents a "snapshot" of the current state of the MC68k microprocessor and its relation to the user's program. It also contains information on the current display mode.
    a. Register values: Two arrays of 32-bit integers, one each for data and address registers.
    b. Program counter: A pointer to the address of the next user instruction to be executed.
    c. Status word: For ease of manipulation, an integer is used, even though only the lower order five bits are relevant to the OBVIAS user.
    d. Register save area: A pointer to the register save area used by the trace trap handlers.
    e. User pointer: The user's window into memory.
    f. User link: A flag determining whether the user pointer is to be dynamically linked to an address register.
    g. Memory: An array of long words, this is used to determine if any memory location in the user's memory window has been altered by the previous instruction.

h. Stack: A linked list containing the history of the data sizes of objects pushed onto the user stack.

i. Current pad: A pointer to the beginning of the actual memory in which the user's assembled code resides.

j. Current pad record: A pointer to the header record of the current user workpad.

k. Stack memory: A pointer to the beginning of stack memory.

l. Stack end: A pointer to the end of stack memory.

m. Modification flags: A complete set of flags to denote whether any CPU value has been modified by the previous instruction. These are used in determining whether the CPU display needs to be updated.

2. User program: The most complex of the OBVIAS structures, consisting of a linked chain of workpad header records.

   a. Workpad headers: A record containing identifying and defining information about the user assembly module stored in it.

      1. Name: A string defined by the user naming the pad.

2. Number: The pad's identifying number, used by the system.
3. Workpad: A pointer to the section of actual memory that the assembled machine code will reside in.
4. Start: An integer defining the line number of the source code entry point.
5. Run mode: The run mode (variable, single step, etc.) of the workpad.
6. Display mode: This controls the visual tracing during execution.
7. Complete: A boolean flag denoting whether the pad is completely defined.
8. Code: A linked list of user source code line records.
    a. Number: The line number of the source code line as it appears on the display.
    b. Line Number: The actual number of the source code line.
    c. Relative Address: The address of the source relative to the beginning of the pad.
    d. Code: A string containing the actual text of the source code line.

e. Length: Length, in bytes, of the machine code.

f. Display lines: Length, in lines, of the displayed statement.

g. Sim flag: Determines whether immediate execution will be attempted.

h. Source: Source operand addressing mode.

i. Destination: Destination operand addressing mode.

j. Identifiers: List of boolean flags denoting whether the code line contains a label, opcode, operands, comment or pseudo operation.

k. Data: A variant record which will contain either the actual machine code of the line, or a list of data if the code lines has the DATA pseudo instruction.

9. Symbol table: A linked list representing the pad's local symbol table.

   1. Name: A string containing the name of the symbol.

   2. Defined: Flag denoting whether the symbol is currently defined.

3. Value: Either the absolute or relative value of the symbol.
4. Kind: Denotes whether symbol is absolute or relative.
5. Where defined: The line number of the symbol definition within the pad.
6. Where used: A list containing the line numbers within the pad where the symbol is used.

3. Windows: Contains all of the OBVIAS window control records.
4. Defaults: Contains system parameters that may be stored to a default startup file.
   a. Delay: Execution speed variable.
   b. Split mode: Flag determining hexadecimal/octal or normal hexadecimal display of machine code.
   c. Bell toggle: Flag denoting whether bell will sound.
   d. Run mode: Default run mode.
   e. Display mode: Default display mode.
   f. Start flag: True is source code file to be loaded during system initialization.
   g. Start file: File name of source code file to be loaded during initialization.

h. Sim toggle: Determines whether immediate execution of code will be attempted.

i. Entry base: Default numeric entry base.

j. Stack size: Size of user stack.

k. Pad size: Size of initial workpad.

l. CPU state: All register values, status word and CPU display modes.

5. Globals: Besides containing all of the default parameters with the exception of CPU state, globals contains current system variables.

   a. Addressing modes: Text strings for addressing mode display.

   b. Breakpoint: Flag denoting that breakpoints are set.

   c. Sim screen: Denotes whether OBVIAS or I/O screen is currently displayed.

   d. Journal: Flag denoting whether journalization is currently in effect.

   e. Global symbol table: A linked list comprising the global symbol table.

      1. Name: The name of the symbol.

      2. Defined: Whether it is defined.

      3. Value: The absolute value of the symbol.

      4. Pad defined: The number of the workpad where the symbol is defined.

5. Where defined: The line number where the symbol is defined.
6. Where global: The line number where the EXTERN or GLOBAL declaration was made.
7. Where used: A linked list of workpad and line numbers where the symbol is used.

6. Help: An array of strings containing the short help messages that are always resident.
7. Error: An array of strings containing the short error messages that are always resident.

The transient or local data structures and variables are, in general, self explanatory. Two exceptions are the pointer variables CURRLINE and NEXTLINE. Both are used by the visual execution unit. Together, they constitute a program counter into the user's code text. While the CPU's program counter always points to the actual machine instruction in memory, NEXTLINE always points to the corresponding text code line. CURRLINE is active immediately prior to execution, pointing to the fully highlighted code line.

Overall, the definition of OBVIAS' data structures proceeded smoothly once the decision to package them was made.

7. Conclusion

The true measure of the success of OBVIAS will not be known until Fall 1984, when the system will be integrated into the CS 400 laboratory. But initial responses have been extremely positive.

A partially functioning version of OBVIAS was demonstrated at the National Educational Computer Conference in Dayton, Ohio in June, 1984. It was well received by dozens of computer scientists who stopped to see it in operation. Among the comments by those who viewed the demonstration were "best thing in the show," "finest debugging tool I have ever seen," and "I wish we had something like that when I learned assembly language."

OBVIAS has far surpassed original expectations, but it certainly is not an end in itself. OBVIAS is merely one step in a continuing trend towards harnessing the power of the computer to free the human mind for more productive work. Design is currently under way at the University of Kansas for Pascal and Fortran environments that will provide similar assistance to students learning those concepts in computer science.

OBVIAS is certainly not perfect. In particular, some of the edit/assemble specifications have not yet been fulfilled. OBVIAS was not intended to impose any limitations on the assembly language syntax, but Version 1.0 does not allow expressions in the operand fields.

OBVIAS, however, will never be truly finished so long as there is a need for it. Like any software system, OBVIAS must evolve to meet the needs of its users, or it will die. Its future success will rely almost entirely on its maintenance.

References

[A] Corvus Systems, Pascal Users Manual, Corvus Systems, San Jose, California, 1984.

[B] Lee, Martha B. "Computer-assisted instruction on and about a personal computer," unpublished Masters Research Paper, Computer Science Department, University of Kansas, 1971.

[C] Schweppe, Earl J. "Dynamic Instructional Models of Computer Organizations and Programming Languages." ACM SIGCSE Bulletin, Vol. 5, No.1 (Febuary 1973), pp. 236-248.

[D] Huebner, Paul F., Skelton, Daniel T., Schweppe, Earl J. "Interactive Instruction Simulation On And Of The Datapoint 2200 Computer." Proceedings of ACM73 Annual Conference, Atlanta, Georgia, 1973 August 27-29, pp. 304-308.

[E] Corvus Systems, Corvus Concept Personal Workstation, Corvus Systems, San Jose, California, 1984.

[F] Corvus Systems, Operating System Reference Manual, Corvus Systems, San Jose, California, 1984.

[G] Ghezzi, Carlo, Jazayeri, Mehdi, Programming Language Concepts, John Wiley & Sons, New York, New York, 1982.

[H] Hopcroft, John E., Ullman, Jeffery D. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Menlo Park, California, 1979.

[I] Aho, Alfred V., Ullman, Jeffery D. Principles of Compiler Design. Addison-Wesley, Menlo Park, California, 1979.

[J] A proposed ISO Standard for Pascal, Pascal News 20, December 1980.

[K] Roehl, Eldon "Visual Emulation of a PDP-11 Computer in Assembly Language Form," unpublished Masters Thesis, Computer Science Department, University of Kansas, 1982.

[L] Corvus Systems, MACSBUG User's Manual, Corvus Systems, San Jose, California, 1984.

[K] Cheng, Mary Owens, "A Visual Interactive Transition Pair Processor," unpublished Masters Research Work, Computer Science Department, University of Kansas, 1976.

Appendix I

The following pages contain actual screen images representing the various sections of OBVIAS.

The images are presented in an order representing a typical OBVIAS programming session. First, the EDIT/ASSEMBLE module is entered. There, program entry is represented by several sample help messages corresponding to code entry.

The remainder of the screen images show the EXECUTE function keys and several running programs.

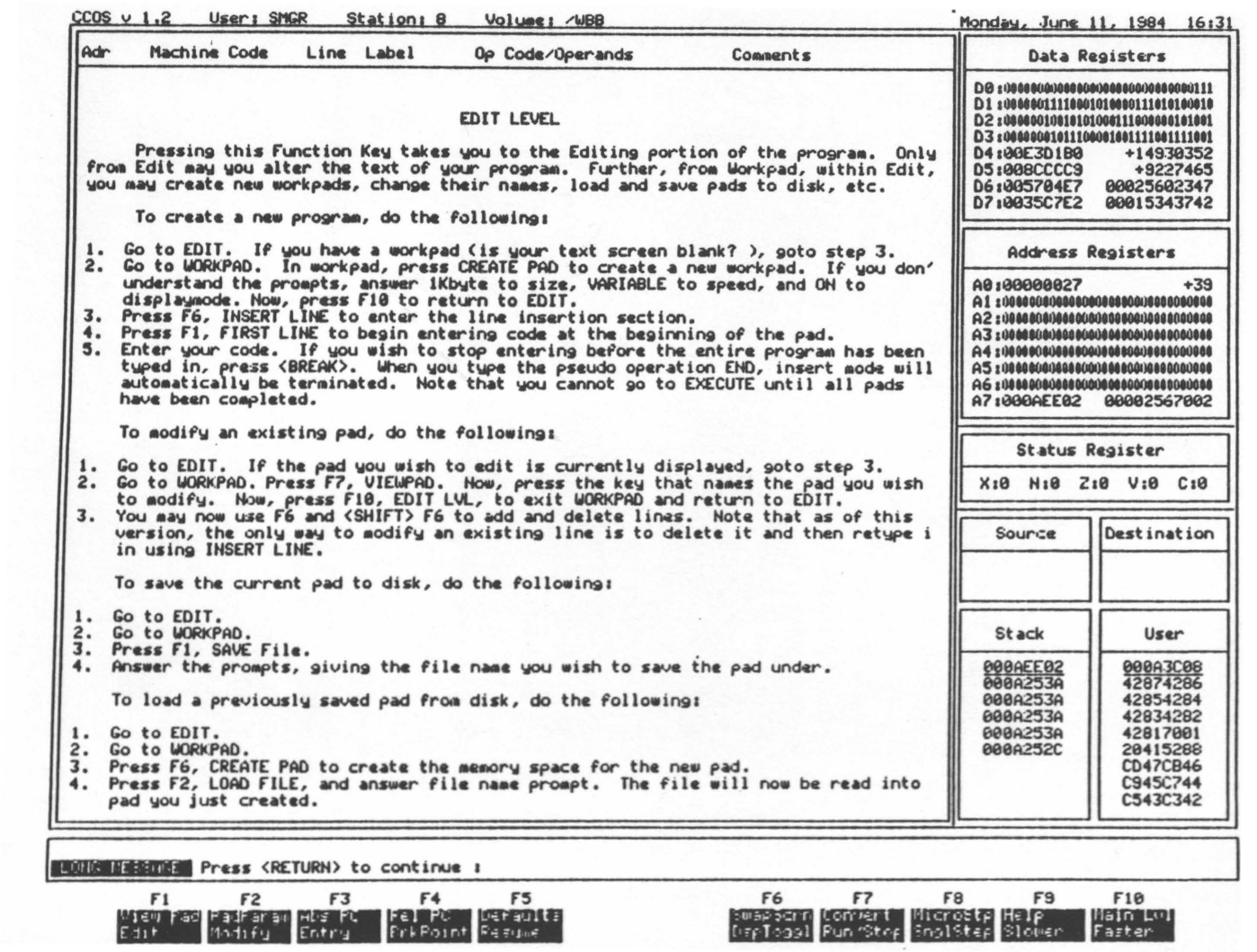


Figure I.1 - Edit level extended help message.

Figure I.2 - Edit level function keys.

```
CCOS v 1.2    User: SMGR    Station: 8    Volume: /WB8                Monday, June 11, 1984  16:31

Adr    Machine Code    Line  Label     Op Code/Operands          Comments

                                    LABEL

     A label is a string of 1 to 8 characters that is used by the programmer to
symbolically define a relative or absolute address, or a numeric constant.

     A label consists of a letter (A..Z) or special character ( '%' or '_' ),
followed by up to seven letters, special characters or digits ( 0..9 ). Further,
labels are classified as absolute or relative.  A relative label represents an
address in your program that will change, depending on the location in which it is
loaded.  An absolute label refers either to a fixed location in memory, or a numeric
constant.

     A label may appear in your program in two manners - definition or use as an
operand.  To have received this message, you are in the process of defining a label.

     To define a label, you must first enter it in the label field of the code line.
If you wish it to be a relative label representing the current code line, you need do
nothing further - the definition is complete.  To define an absolute label or a
relative label representing some other point in the program, you must follow the
label with the EQU pseudoop in the opcode field, followed by desired operands.

     Absolute label definitions:

ASCIIZ    EQU     'Z'         ; 'ASCIIZ' refers to ascii value of char 'Z'.
WARMBOOT  EQU     4           ; 'WARMBOOT' refers to absolute address 4.

     Relative label definitions:

GOBACK    ADDQ.B  #1,D0       ; 'GOBACK' refers to this code line
GOOBACK   EQU     GOBACK - 4  ; 'GOOBACK' refers to relative address 'GOBACK' - 4.

     If you use an expression to define a label with EQU (as in previous example),
the defined label is ABSOLUTE iff the expression is:

          RELATIVE - RELATIVE
          ABSOLUTE +/- ABSOLUTE
          ABSOLUTE * ABSOLUTE
          ASBOLUTE / ABSOLUTE

     The defined label is classified as RELATIVE iff the expression is:

          RELATIVE +/- ABSOLUTE
          ABSOLUTE + RELATIVE

                    Data Registers
D0:00001111111110000000110000111000
D1:00001001110111101000110101101101
D2:00000110000110010111111011001011
D3:00000011110001010000111010100010
D4:00000010010101000111000000101001
D5:00000001011100001001111001111001
D6:00000000111000111101000110110000
D7:00000000100011001100110011001001

                   Address Registers
A0:0000002A                    +42
A1:00000000000000000000000000000000
A2:00000000000000000000000000000000
A3:00000000000000000000000000000000
A4:00000000000000000000000000000000
A5:00000000000000000000000000000000
A6:00000000000000000000000000000000
A7:000AEE02   00002567002

                   Status Register
X:0  N:0  Z:0  V:0  C:0

        Source          Destination

        Stack           User
000AEE02        000A3C08
000A253A        42874286
000A253A        42854284
000A253A        42834282
000A253A        42817001
000A252C        20415288
                CD47CB46
                C945C744
                C543C342

LOAD MESSAGE  Press <RETURN> to continue :
```

Figure I.3 - Code entry help message (Label Field).

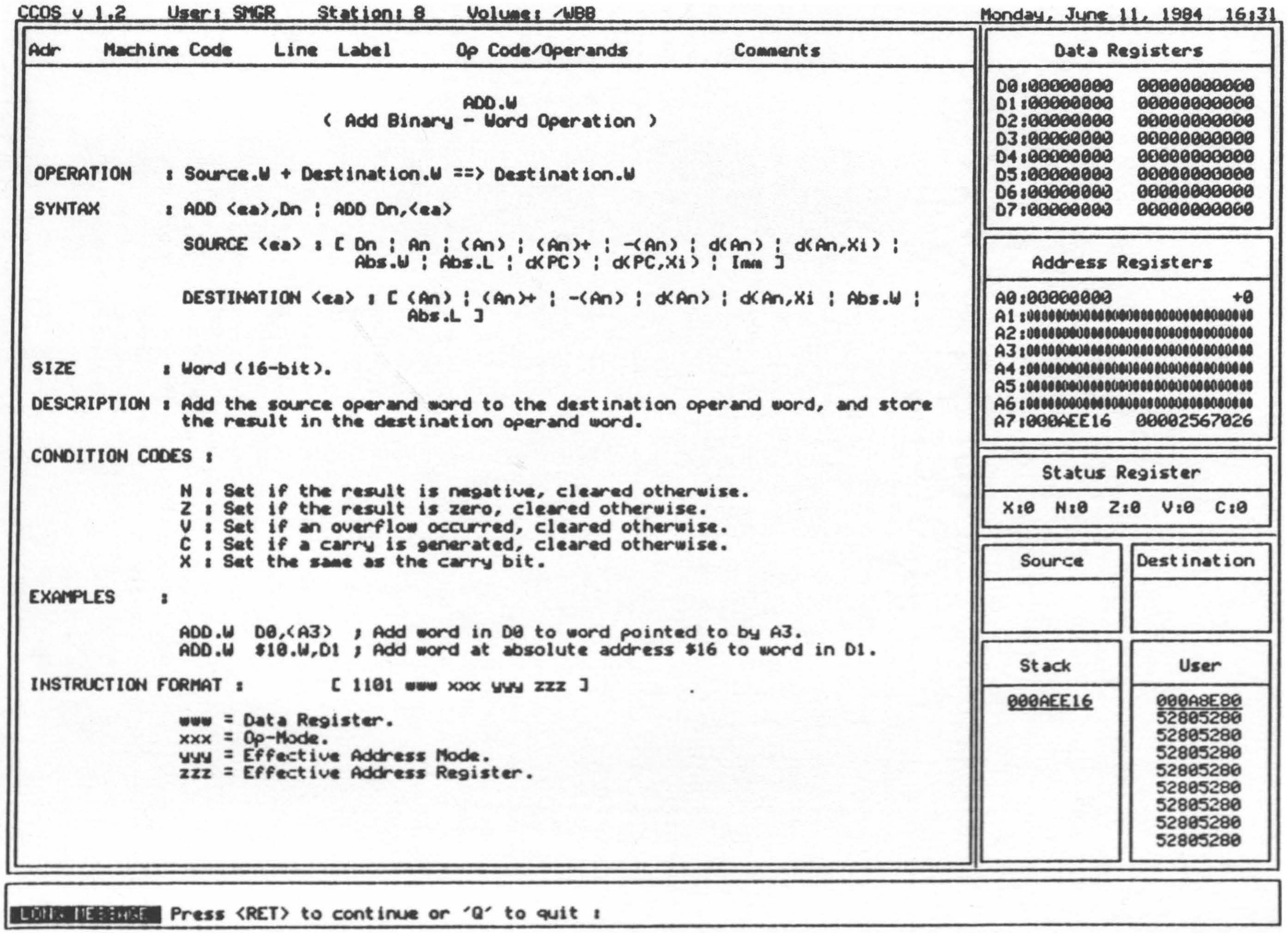


Figure I.4 - Code entry help message (Opcode field).

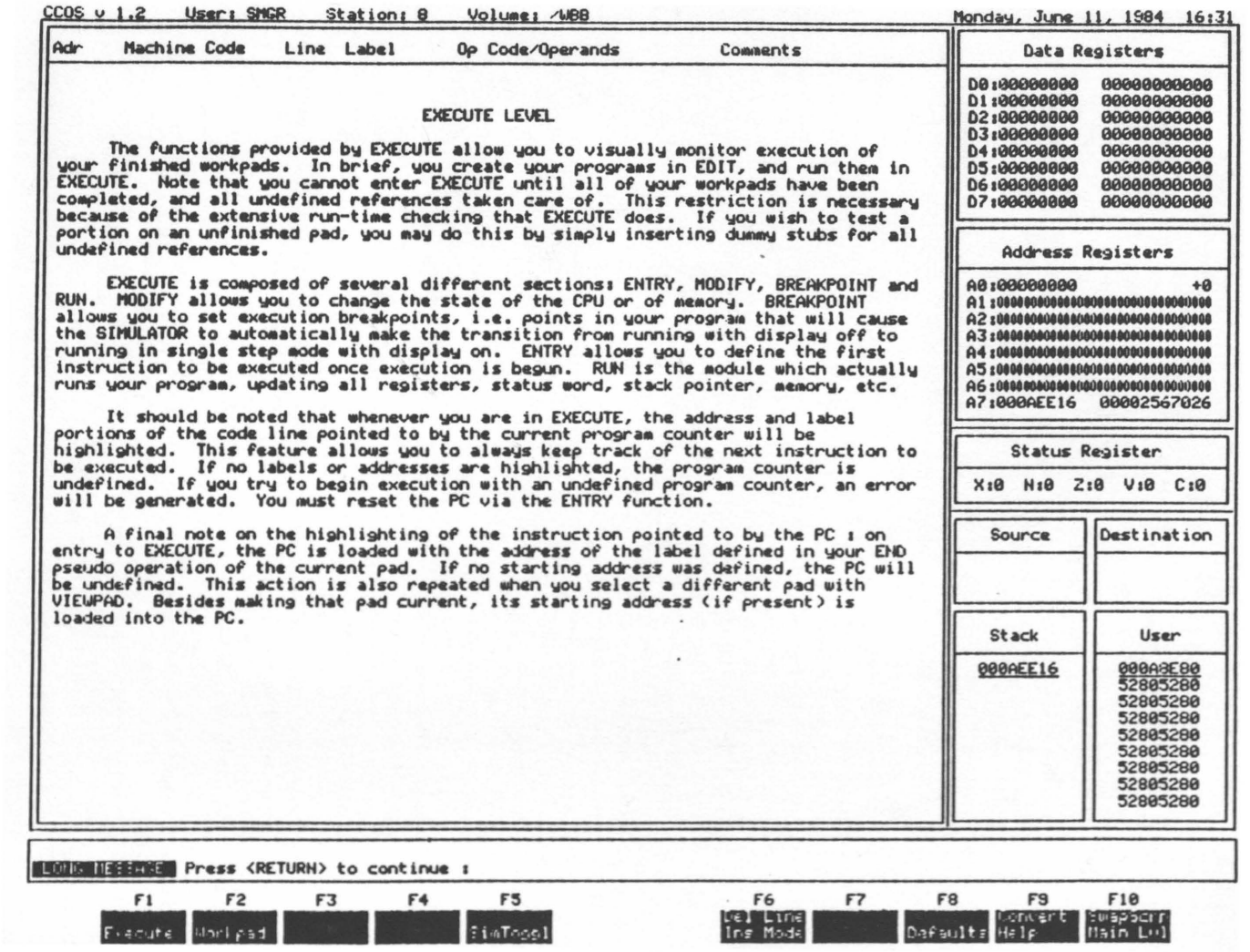


Figure I.5 - Execute level extended help message.

Figure I.6 - Execute level function keys.

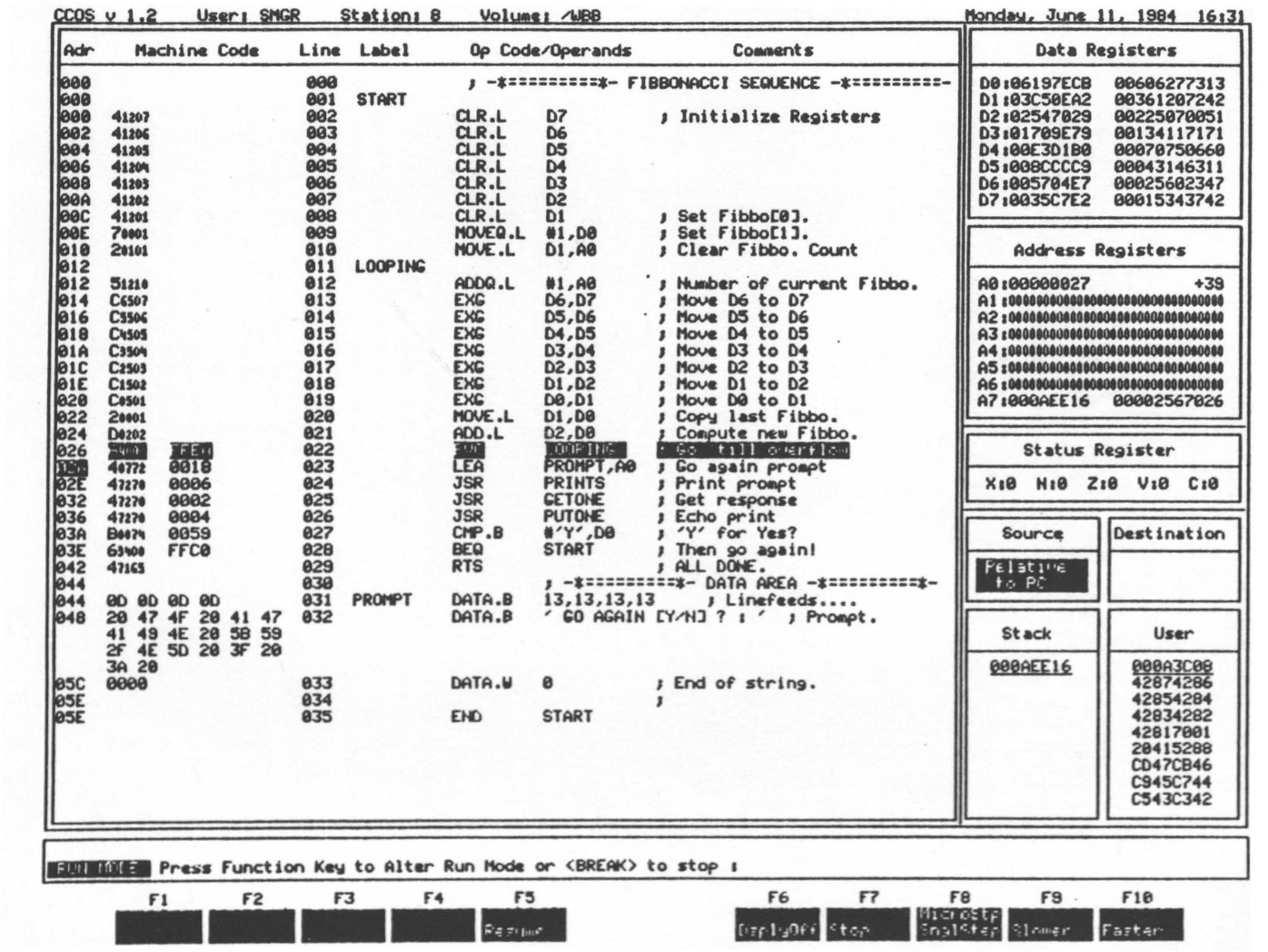


Figure I.7 - Running program, variable speed.

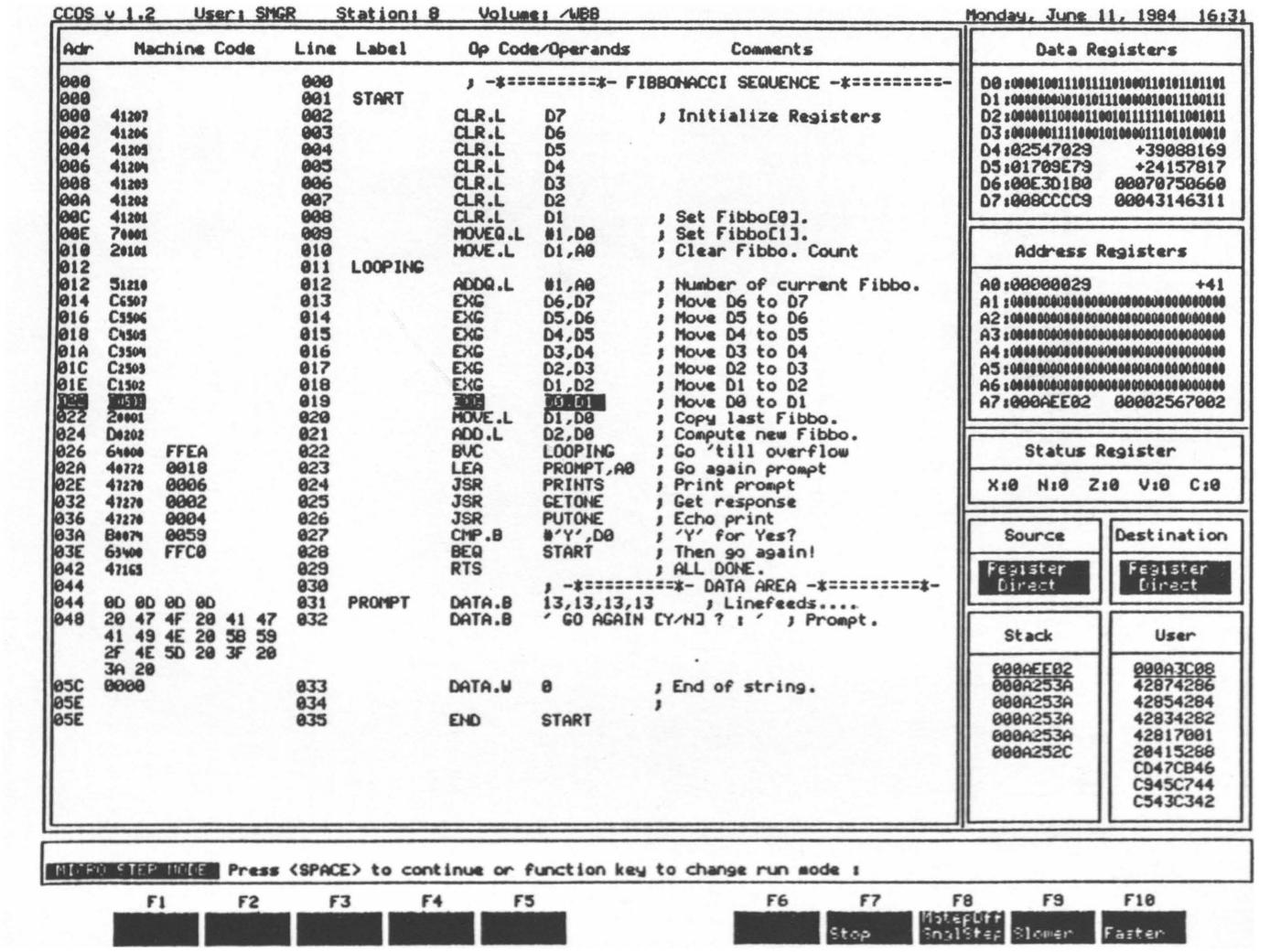


Figure I.8 - Running program, Micro Step mode.

Figure I.9 - Running program, stack operation.

Figure I.10 - Running program, scrolling demonstration.