01 Jun 1984

A Graphical Representation of an Executing Program

Sherry A. Lile

Arlan R. Dekock
Missouri University of Science and Technology, adekock@mst.edu

John Bruce Prater
Missouri University of Science and Technology

Darrow Finch Dawson
Missouri University of Science and Technology

A GRAPHICAL REPRESENTATION OF AN EXECUTING PROGRAM

Sherry A. Lile*, Arlan R. DeKock,
John B. Prater, and Darrow F. Dawson

CSc-84-9

Department of Computer Science

University of Missouri-Rolla

Rolla, MO 65401 (314) 341-4491

*This report is substantially the M.S. thesis of the first author, completed July, 1984.

ABSTRACT

This thesis describes the rationale for a computer program used as a teaching aid, as well as, the design, development, and implementation of that computer program. The program is a prototype that displays, line by line, a graphical depiction of a BASIC program being executed. Icons are used to represent FOR-NEXT loops, the contents of variables, I/O, and other programming elements. The internal logic associated with the Arithmetic-Logic Unit is also represented. The purpose is to produce an environment where the student may visualize the processes occurring in memory by viewing a symbolic portrayal of an executing program. The example program is at an introductory- to intermediate-level, designed to be a review for the novice BASIC programming student.

ACKNOWLEDGEMENT

Sincere appreciation is extended to Dr. Howard D. Pyron and his BASIC students for their time and cooperation.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I. INTRODUCTION

A. PROBLEM DEFINITION

Students of computer programming, like students in general, may have difficulty understanding basic concepts. In computer programming, one area of consistent confusion pertains to understanding the processes occurring in memory during the execution of a program. It is difficult to visualize the abstract internal workings of computer memory. Instructors try to demonstrate these processes symbolically at the blackboard. They employ a set of commonly used icons, or figures, to illustrate the contents of memory and the computations of the Arithmetic-Logic Unit while teaching. The research described in this paper was undertaken to develop a teaching aid that would depict these processes and to test the feasibility of automating the icon representation.

B. PROBLEM SIGNIFICANCE

The problem of misunderstanding fundamentals is universal. Trowbridge and Bork have noted that

> "... students display great commonality in the processes whereby they learn new material. Often they go through the same steps and encounter the same pitfalls along the way. Experienced teachers in every field share the observation that students repeatedly misunderstand important concepts in the same way, and predictably display certain preconceptions which are impediments to learning the crucial ideas of that field." [TROW81a, HAWK78, MCCL80, TROW81b, TROW80, TROW81c]

The concepts associated with variables are fundamental within the architecture of the traditional type of computer.

The student must be able to comprehend how a value is stored, or contained under a label in the memory component of a computer. It is the value of this variable upon which the Arithmetic-Logic Unit will act.

> "The concept of variables and their values is computer programming's most basic concept. . . . To understand the learning of computers [sic], it is absolutely essential to keep the concept of variables in mind." [DAVI82]

That the basic concept of variables is difficult to comprehend was confirmed in a study by Lance Miller. In 1981, Miller studied how students expressed procedural specifications in English for file searching problems. He observed that the students had difficulty in being explicit about procedures and assumed that variable references were clear from the context of their paragraph. [MILL81]

C. PROBLEM SOLUTION

A possible solution for the problem that students have in understanding basic programming concepts may be found in a presentation of a pictorial representation of the important processes occurring inside the machine while a program is executing. Commonly, a person states, "I see." when a concept is comprehended. In this sense, seeing is associated with comprehension. When results are doubted, verification is required, and the person says of the questionable material, "Let me see." In both instances--learning and confirming--seeing is at least part of the solution.

It is virtually common knowledge that pictorial communication is both more effective and more appealing than script or numeric displays. A study by Tullis [TULL81] compared graphic display formats (both color and black-and-white) with narrative and tabular formats. The formats were viewed by test subjects. Researchers then examined the subjects' speed in interpreting displayed data and requested the subjects' format preference in daily work. Tullis found the speed at which data was interpreted was significantly faster for the graphic formats than for the narrative and tabular formats. The subjects also rated a clear preference for the graphic formats.

Pictorial communication is effective in many ways. It allows a large quantity of data to be transferred more readily. Further emphasis may be given to a dynamic process by highlighting or blinking part of the picture to draw special attention to the relevant result. Displayed information can be assimilated quickly by the viewer. It is the working hypothesis of this project that a pictorial representation of an executing program will help the student comprehend what the program is doing. The following is a description of an effort to create a programming environment incorporating these graphical, real-time displays.

II. LITERATURE REVIEW

For students to learn a subject without any misunderstandings is certainly desirable. Misunderstanding a programming concept normally results in programs that do not execute correctly. Since the program error reflects less than complete understanding, the novice is dependent upon the machine and, for the most part, helpless in trying to debug the program.

Efforts have been made to alleviate this problem of interfacing with the computer. Two approaches use the machine itself--debuggers and tutorials. Debuggers help by showing where and when the values change. They do not say why a variable has a change of value. Tutorials explain how and why variables may change values. However, tutorials rarely show why a student's program does not work.

This portion of the paper will survey five debuggers and three tutorials. The purpose of the project is not to develop a debugger nor to write a tutorial. The aim is to examine what is presently used and design a method that would incorporate the desired characteristics from either of these allied areas. The discussion of the debuggers will include a description of their processing environment, mode of access, and capabilities. The three tutorials that will be reviewed each present instruction for a programming language.

MANTIS [ASHB73] is an interactive debugging facility for FORTRAN. It was designed to be an integral component of a timesharing system at the University of Oregon. In a timesharing environment, programmers may enter code, correct typing errors, compile to find syntax errors, and make test runs to modify logical errors. Thus, in a short period of time an application program could be made a finished product, providing the user can locate the source of and correct any errors.

MANTIS may be accessed at any time; it does not require recompilation. A user need only issue a recognized MANTIS command to utilize this debugger.

MANTIS allows the user to set breakpoints and to trace variables and arguments in the programs to be debugged. Breakpoints allow the user to specify executable program locations where the debugger gains control at time of execution. Breakpoints in MANTIS may be set on subroutine calls and/or returns. A trace will display the value of the indicated variable or expression as it changes throughout the program. MANTIS will also assist the user by allowing the values of variables and arguments to be changed at breakpoints, by permitting the initiation of program execution at any point in the program, and by allowing the alteration of the normal execution pattern of statements within programs.

A single debugger that could service PL/I, BASIC, and FORTRAN was implemented in an interactive environment at Dartmouth College [ELLI82]. A single debugger to service all three languages is especially useful since programs written in any language may call procedures written in any other language. However, the users must invoke the Dartmouth debugger--prior to executing their programs--in order to avoid recompilation and re-execution of the programs. The debugger lets users trace execution of their programs, set breakpoints, and view and change variables' values. Breaks in execution may be caused every time a certain statement is encountered, if one or more variables change their values or, if user-defined conditions become true.

A third interactive system was developed to serve as a translator to support run-time debugging [JOHN79]. It was designed by M.S. Johnson to be language-independent. This debugging system may be viewed as a routine which is invoked during execution of a compiled object program. (Typically, the compiler reads the source program from a file, allows no update to that source during the compilation process and produces an object program that is close to machine language. A loader then combines this object program with other objects and with standard library routines.) This debugger provides a single run-time debugging interface, which allows users to debug programs written in various high-level source languages. This system reports errors and displays diagnostic information interactively.

Another system, the Advanced Interactive Debugging System, AIDS, was designed for batch or interactive mode [HART79]. AIDS may be initiated before the program's execution, or it automatically gains control and solicits commands at the detection of a hardware error. AIDS does not require prior invocation, program recompilation, recombining load modules, or re-execution.

AIDS had the following design considerations: symbolic and non-symbolic (i.e. machine addresses) referencing; a uniform way to debug programs written in any supported language; data display; and program tracing. AIDS may be used to reference internal and external symbols for variables, labels, lines, and entry points. AIDS allows the programmer to display or to change the contents of data items and memory locations. Execution of commands within a debugger procedure allows branching to a specified label, whether in previous or following code. AIDS has a limited text-editing capability and a help facility.

Debuggers allow programmers to look at the internal states of the machine at selected points. However, the debuggers do not furnish any hints or information concerning how the machine arrived at its present state. Tutorials, on the other hand, provide explanation. Tutorials present the correct programming concepts and promote student understanding.

A computer tutorial on a programming language may be very beneficial. Some tutorials emphasize analyzing student reasoning. Other tutorials employ techniques to make computer-aided instruction, CAI, more like human instruction.

BIP [WEST77], a CAI programming tutor, was designed and built to be a self-contained, full course in presenting the programming language BASIC. It features a sophisticated technique for deciding what material should be presented to the student. It also has excellent graphic displays. BIP analyzes students' programs by running them on test data.

Another example of a CAI programming tutor is SPADE-0 [MILL79]. This program is designed to teach students to write simple LOGO programs. With this tutorial the student is required to perform the programming processes of design, coding, and debugging. The student must justify the code before being allowed to enter it.

MENO-II is a computer-based tutor designed to help novice Pascal programmers in conjunction with a lecture course [SOLO83]. It has two major components: the BUG-FINDER, to catch run-time bugs in student programs, and the TUTOR, to instruct the student about possible misconceptions responsible for the error. The BUG-FINDER recognized two types of errors; problem-independent ones (semantic bugs) and problem-dependent ones (pragmatic bugs). It could access a database of 18 common bug types. After finding a bug it passes information to the TUTOR. The TUTOR hypothesizes [sic] potential misconceptions that the student might have had that may have led to the programming bug.

Debuggers are valuable programming aids. But they supply no explanation. The user cannot see how the program arrived at the unexpected results. Users may employ traces and set breakpoints at arbitrary program locations and examine the internal states of the machine. Although setting a breakpoint interactively is a step above batch operation, the programmer must know where to position the break and on what to trace. It is also possible that a student's program will execute without a detected error, but it may produce the wrong answers because of logical errors.

Tutorials are excellent and necessary teaching aids. They cover topics and furnish examples. But tutorials do not show what happens to a particular program in execution. A teaching aid that shows a step-by-step execution in a format that is appropriate and easy to understand should be helpful. It would explain, by portraying in detail, what was occurring in the execution of a program. The purpose of this thesis is to evolve and evaluate a teaching aid to do that--to display, step by step, a graphical depiction of an executing program.

III. PROCEDURE

A. DESIGN AND DEVELOPMENT

As was noted, it is common for novices in any field to misunderstand fundamentals. The desire to address this situation in programming is the motivation for this project. The hypothesis is: If the major internal computation actions have a corresponding external icon, the students' understanding of computation will improve.

The following concepts are considered fundamental to programming and are incorporated into the example program.

* Variables
* Memory
* Arithmetic-Logic Unit
* Assignment
* Input/Output
* Looping
* Arrays
* Arithmetic

These important concepts are always covered in introductory courses in programming. Each of these concepts is discussed in turn, together with the way in which values and operations are identified in the step-by-step execution of the program in the prototype system.

* Variables

Variables are memory locations with names that store values. Those values can be changed by performing operations that involve the variables. Names are given to storage locations to identify them uniquely; then a storage location may be assigned a value. Variables may be of different types; i.e., they may be associated with different types of data. Two such types of data are numeric and character. For this project, a variable in memory is represented by the variable name being displayed adjacent to a box. The box represents the memory location at which the value is stored.

A close connection exists between the variable name and its value. This is illustrated in the following: when a variable is referenced in the program, the variable and value are highlighted in the memory portion of the display.

* Memory and the Arithmetic-Logic Unit

Memory and the Arithmetic-Logic Unit (ALU) are two distinct but interdependent components of the von Neumann-type computer. The variable name identifies the memory location whose content is to be transferred to the ALU. Representation should be different when a variable or an operand is depicted in the ALU. No variable names are used in that component of the computer. Registers in the ALU are not reserved for particular variables. It is the value that is transferred upon which the ALU acts. No name or label is retained by the value when it enters the ALU.

The two different uses of a value, ALU versus memory, are reflected in the two distinct ways they are graphically depicted during execution. In the ALU section, only the value being emphasized is highlighted, whereas in the memory section both the value and its variable name are highlighted.

* Assignment

The Assignment concept is perhaps the most basic and the simplest. A value needs a variable name in memory to accommodate it when it is not being used as an operand in the ALU. Its identity is the variable name that is associated with it. In an Assignment statement a value on the right-hand side of the assignment symbol is assigned to a variable on the left-hand side. During execution of an Assignment statement, the memory section highlights the variable name from the left-hand side of the assignment symbol and flashes the value being assigned to it for more emphasis.

* Input/Output

Every program with purpose performs I/O on data. The Input portion of a program supplies data and offers flexibility. Output yields the results: the solution for a problem or verification that the program's purpose is accomplished.

The Input and Output programming concepts each has two aspects to be displayed. The first aspect of Input identifies the incoming data by highlighting it. The second aspect of Input is that of an indirect Assignment. During the Assignment, attention needs to be focused on the value as it is stored in the correct memory location associated with the indicated variable name.

Similar to Input, Output has two aspects to be shown. However, depicting an Output action reverses the emphasis displayed by the Input. The first aspect of Output identifies the outgoing value and its variable name by highlighting them in the memory area. The second aspect is the resulting display in the Output area. More emphasis is given to the Output area by displaying the output in reverse video and flashing it.

* Looping

The concept of looping is important because loops are an efficient way to handle repetition. There are three components involved in the mechanics of a loop. The first component initializes the index variable. This initialization is accomplished through an Assignment statement in the memory section. The second component increments the index variable. This phase of looping occurs in the ALU section. The third component compares the index variable with the terminal or final value of the loop, and also occurs in the ALU section.

In the second component, the value of the index variable is increased by a designated increment after the loop is executed. The increment is demonstrated by the value of the increment being added to the value of the loop variable. Since only the values are shown in the ALU section, the variables and their values are highlighted in the memory section. The values in the ALU section are shown in reverse video to give them more emphasis. The sum of the addition is portrayed in reverse video and is flashing for added emphasis.

The third component shows the sum being compared with the terminal value of the loop. The comparison tests if the value of the index variable has exceeded the terminal value. This test is shown in reverse video in the ALU section. The answer is shown in reverse video and flashing to give it greater significance. The test for completion of the loop is performed before the body of the loop is executed. Thus, if the initial value of the index variable is greater than the terminal value, the loop will not be executed. Execution would then continue with the statement following the end of the loop. However, if the initial value of the index variable does not exceed the terminal value, the statements within the body of the loop are executed.

* Arrays

Arrays are used to store homogeneous data. An array is a data structure that was developed for the purpose of grouping like data under the same name with reference to individual members provided by use of subscript values. In this graphical representation, an array is indicated by juxtapositioned boxes. A two-dimensional array is depicted similarly. The elements are presented contiguously in row-major order with the appropriate subscripts under the boxes. This is the case for both numeric and string variables.

* Arithmetic

The Arithmetic statement includes an Assignment and an arithmetic expression. Following the precedence of operators, the expression is simplified using two operands and their joining operator. The operator of highest precedence and its two operands are highlighted in the source text. If an operand is a variable, then the variable and its value are emphasized in reverse video in the memory section. The operator and the values of the operands are depicted in reverse video in the ALU section. The result of the operation flashes to give it the most attention. This procedure is followed, one operation at a time, until a single value remains on the right side of the assignment symbol. This value is then assigned to the variable on the left side of the assignment symbol.

B. IMPLEMENTATION

Two possible approaches for implementation are

1. to modify an interpreter, or
2. to create a data file that could be displayed to simulate the desired processes.

The first approach involves changes in the design, coding, and implemention of an interpreter. This approach would be aided considerably if access to and documentation for proprietary code were available. A request for this information was not acknowledged. The second approach is more a "brute force" effort and requires much coding, but at a lesser degree of difficulty than the former.

As this project was viewed as a prototype--a form that might serve as a model for later efforts--it uses a "hard-wired" program to generate the desired presentation. The project tests an idea for a teaching aid; it does not test an operational version of that aid.

The decision to hard-wire the prototype has a significant effect on storage. The example program would be displayed as if each statement were individually executed. A screen would be devoted to each statement or operation of a statement. Each screen to be displayed consists of 20 lines, or records, of 100 bytes each. Thus, one screen contains 2,000 bytes. A program of 90 lines of code could require over 500,000 bytes of disk storage.

A second consequence of hard-wiring the prototype is the delay associated with I/O. The data file would be accessed repeatedly, twenty records at a time. Studies of man-computer interaction [LAY81] have shown that delays can adversely affect human and consequently system performance. Delays are irritating and appear to interrupt concentration.

The BASIC language was selected for the implementation and offers a way to reduce the delay. The additional video memory that is resident to accommodate the Color/Graphic capability is only partially used in text mode. BASIC may employ this additional video memory by writing three additional screens of textual information into the resident video memory. With the four screens in video memory, the switching time among these four screens is instantaneous [CONK83]. However, writing a group of four screens into video memory requires approximately fourteen seconds. For this time period the system is occupied and any student request is queued.

This prototype actualizes the ideas by displaying an example program written in the BASIC language and utilizing a microcomputer. The BASIC language has been enhanced and is widely used. Microcomputers are popular and very appropriate for individualized self-paced study and many include a version of BASIC in their software. The capabilities and availability of BASIC and the microcomputer were determining factors in their selection for this project. It was decided that BASIC would be the language of the example program and that the teaching aid would be available on a microcomputer, in particular, on the IBM PC.

The design requires that the screen be divided into four display areas to properly depict the source program and the previously mentioned concepts. The screen of the monitor is partitioned into four areas, or windows. These areas are labeled SOURCE PROGRAM; MEMORY; LOGIC, for the Arithmetic-Logic Unit; and OUTPUT AREA. The upper left area, the SOURCE PROGRAM window, displays fifteen lines of source program that surround the statement being executed. The upper right area, the MEMORY window, is devoted to iconically representing the contents of memory. The lower right area, the LOGIC window, shows the processes taking place in the Arithmetic-Logic Unit. The lower left window, OUTPUT AREA, is used for the source program's output. The SOURCE PROGRAM window is sixteen lines by forty characters, the MEMORY window is also sixteen by forty, the LOGIC window is six by thirty, and the OUTPUT window is six by fifty. Please see Figure 1.

With so much information being displayed on the screen, it is important to develop aids that help the student see what is changing or taking place. The IBM PC offers three display options to assist in this; blinking, high-intensity, and reverse video. Helping to direct the student's attention, different portions of the screens are displayed in high-intensity, high-intensity with blinking, reverse video, reverse video with blinking, or the normal image of white on black. The figures included in this section will

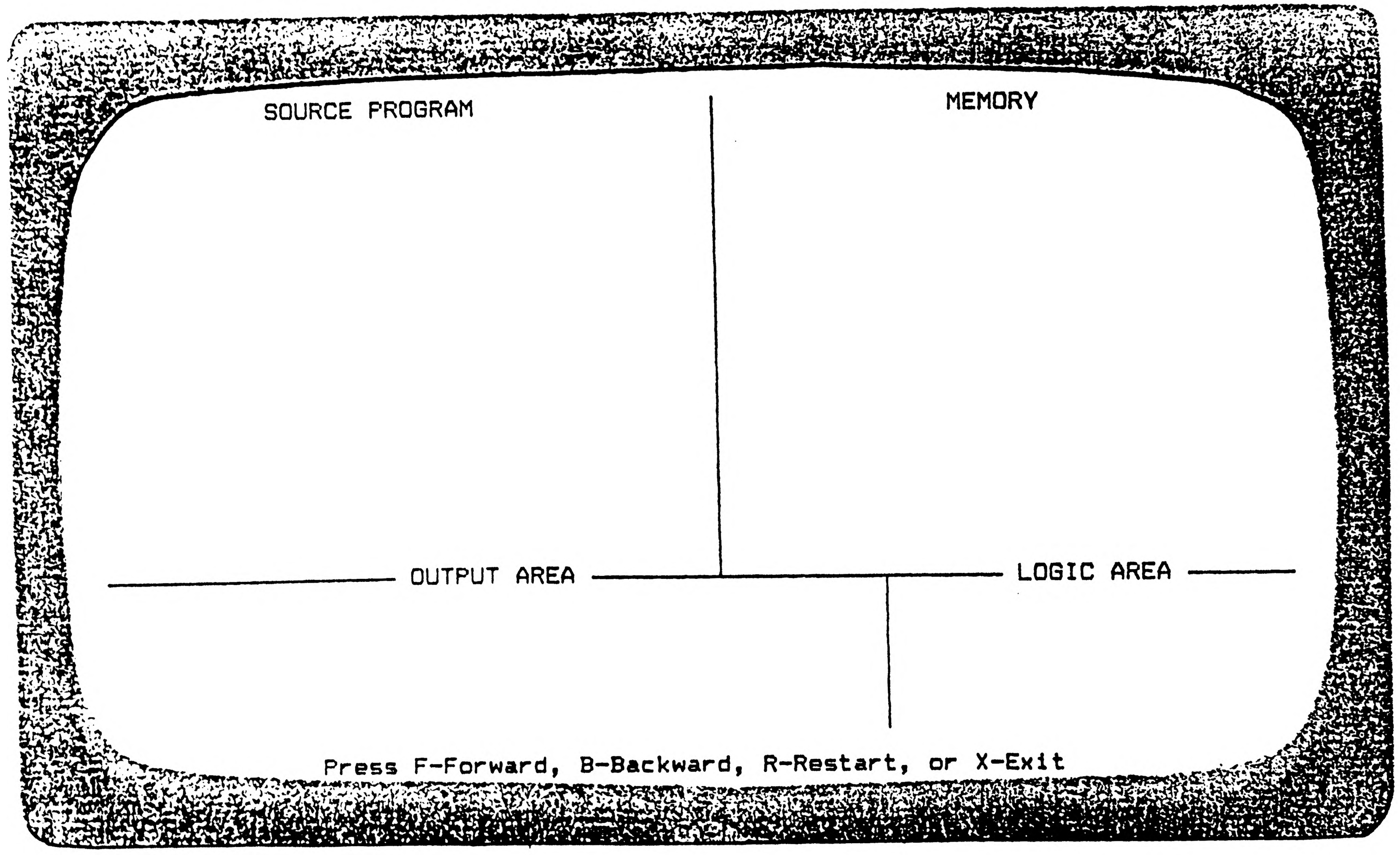


Figure 1. The Screen Partitioned into Display Windows

represent an area displayed in high-intensity on the screen surrounded by a flattened ellipse. Items that are displayed in reverse video are enclosed in a rectangle. Entries that are displayed blinking are ringed in a zig-zag pattern.

The students' attention is to be focused first on the line or part of a line being executed in the SOURCE PROGRAM window. The SOURCE PROGRAM window displays fifteen lines of source code at one time. Attention is focused on the line being executed by a pointer to the left of the line number. To attract the student's attention to the proper statement, the pointer is blinking in high-intensity. The statement being executed also is displayed in high-intensity, but not blinking. Blinking the statement made the screen too busy and somewhat distracting.

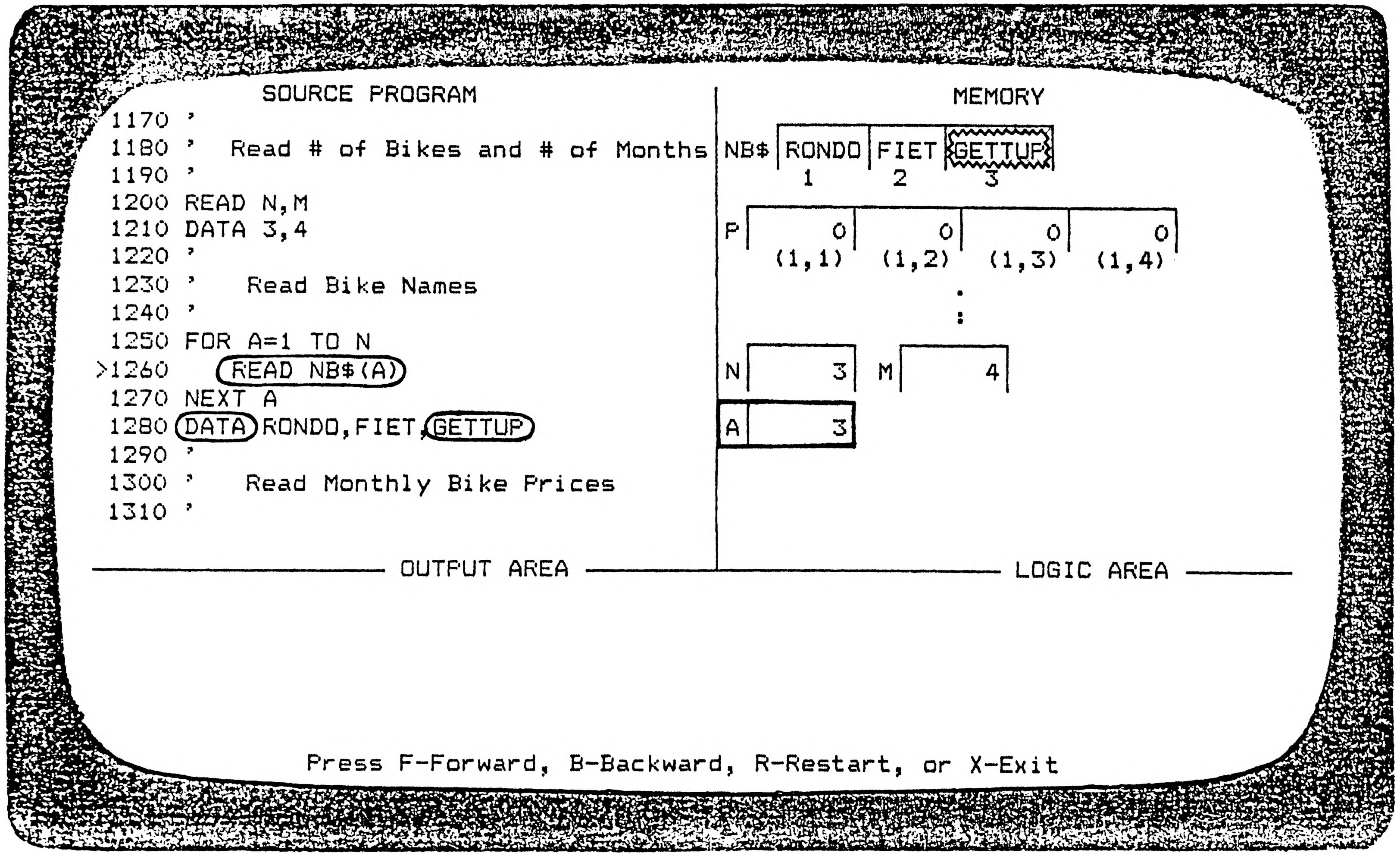
Each statement of the example program in the SOURCE PROGRAM window is executed sequentially. As each individual statement is executed the screen changes, giving the effect of scrolling. Simultaneously, the corresponding changes in memory and the ALU are depicted in the MEMORY and LOGIC windows.

BASIC variables may be either numeric or string. A variable is depicted in MEMORY by the variable name adjacent to the box that stores the value associated with that variable. A reference to a particular variable in the SOURCE PROGRAM causes that variable to be displayed in MEMORY. The variable and its value are depicted in MEMORY in reverse video. The value is also blinking. Unassigned numeric variables are initialized to zero, and unassigned string variables are initialized to null strings. Null is depicted as a box of zero length. When a string variable is assigned a value, the length of the box expands to equal the number of characters in the assigned string value.

The LOGIC window does not display variable names. This conforms to the fact that registers in the ALU are not reserved for particular variables. The student may have difficulty identifying what variables are associated with the values being displayed in the LOGIC window. This situation is remedied by highlighting the relevant values and their variable names in MEMORY.

Input is represented in the example program by the READ/DATA combination. The program was written to have the DATA statement in close proximity to the READ statement in order to aid the student in identifying the correct data item. The READ/DATA combination is depicted in the example program by highlighting its two components on the same screen, when the READ statement is executed. First, the particular data value of the relevant DATA statement is shown in high-intensity in the SOURCE PROGRAM window. Second, the Assignment component is depicted in reverse video with the value blinking in the MEMORY window. In this manner the student is able to see where the value originates and where it is stored. An example of the READ/DATA combination is illustrated in Figure 2.

The same general pattern is used for output directed to the OUTPUT AREA window. The object of a PRINT statement is displayed blinking in reverse video in the OUTPUT AREA. If



SOURCE PROGRAM

```
 1170 '
 1180 '   Read # of Bikes and # of Months
 1190 '
 1200 READ N,M
 1210 DATA 3,4
 1220 '
 1230 '    Read Bike Names
 1240 '
 1250 FOR A=1 TO N
>1260    READ NB$(A)
 1270 NEXT A
 1280 DATA RONDO,FIET,GETTUP
 1290 '
 1300 '    Read Monthly Bike Prices
 1310 '
```

MEMORY

```
NB$ |RONDO|FIET|GETTUP|
       1     2     3

P |     0|     0|     0|     0|
   (1,1)  (1,2)  (1,3)  (1,4)
                 .
                 :
N |     3|  M |     4|

A |     3|
```

OUTPUT AREA

LOGIC AREA

Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 2. READ/DATA Combination for Input

a variable is the object of the PRINT, that variable and its value are highlighted in reverse video in MEMORY.

Illustrating a FOR-NEXT loop requires depicting the logic that transpires in the ALU. The portrayal of a FOR-NEXT loop proceeds as follows:

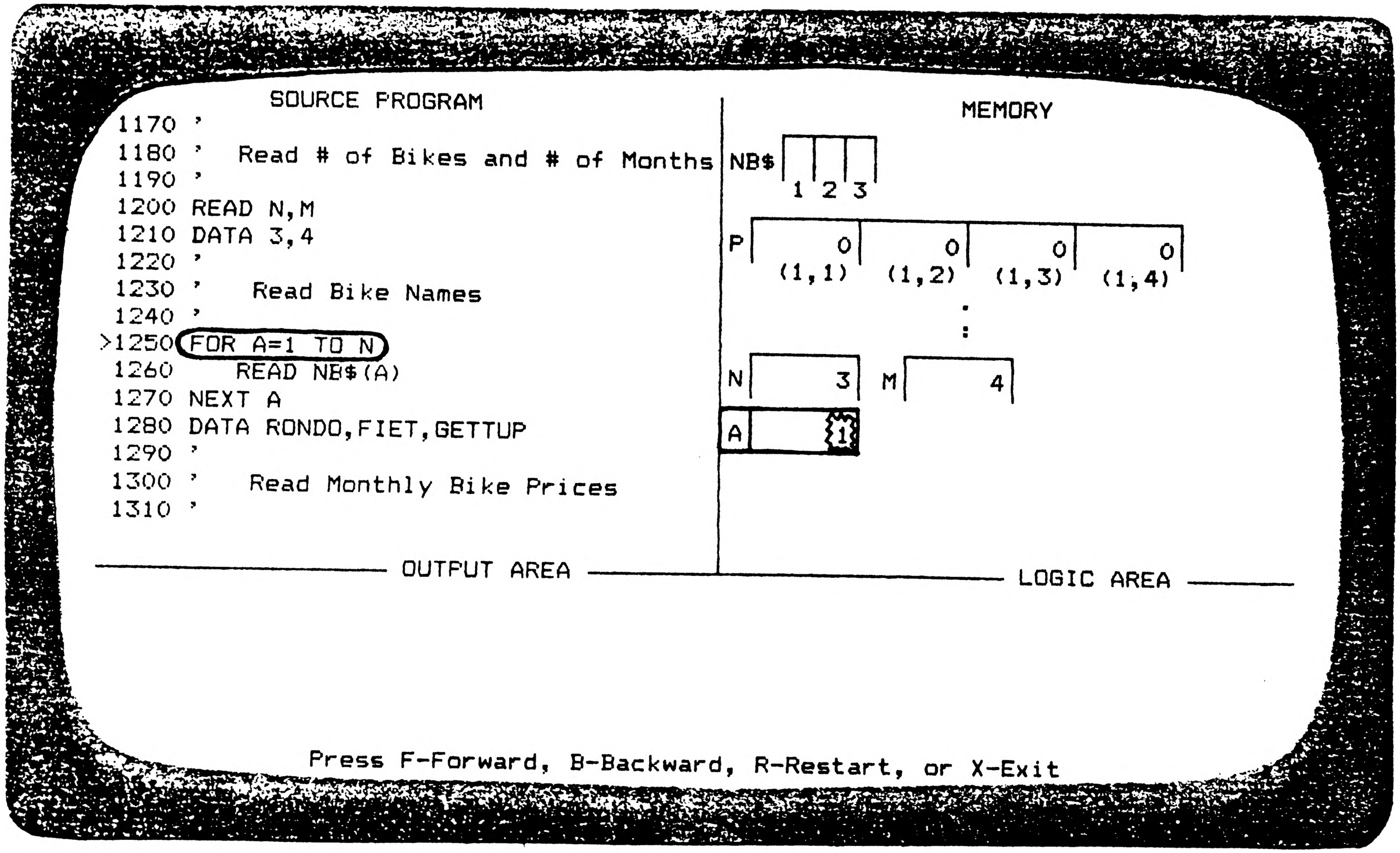
1. The FOR statement is highlighted in the SOURCE PROGRAM window where the statement is shown in high-intensity. The MEMORY window depicts the initial assignment by displaying the index variable in reverse video. The value being assigned is in reverse video and blinking. This is shown in Figure 3.

2. The body of the loop is executed line by line.

3. After the body of the loop, the value of the index variable and the value of the increment are added in the LOGIC window. These values are displayed in reverse video. Their sum is displayed in reverse video and blinking. The same screen shows the index variable and value displayed in reverse video in the MEMORY window. See Figure 4.

4. The next screen shows the sum from the previous operation compared with the terminal value of the loop. The test to check if the sum is greater than the terminal value is in reverse video and the answer to the comparison is in reverse video and blinking. Testing the index variable at this point is inconsistent with the way most versions of BASIC are implemented, although it is consistent with FORTRAN. The correction for this implementation is discussed in Section V.

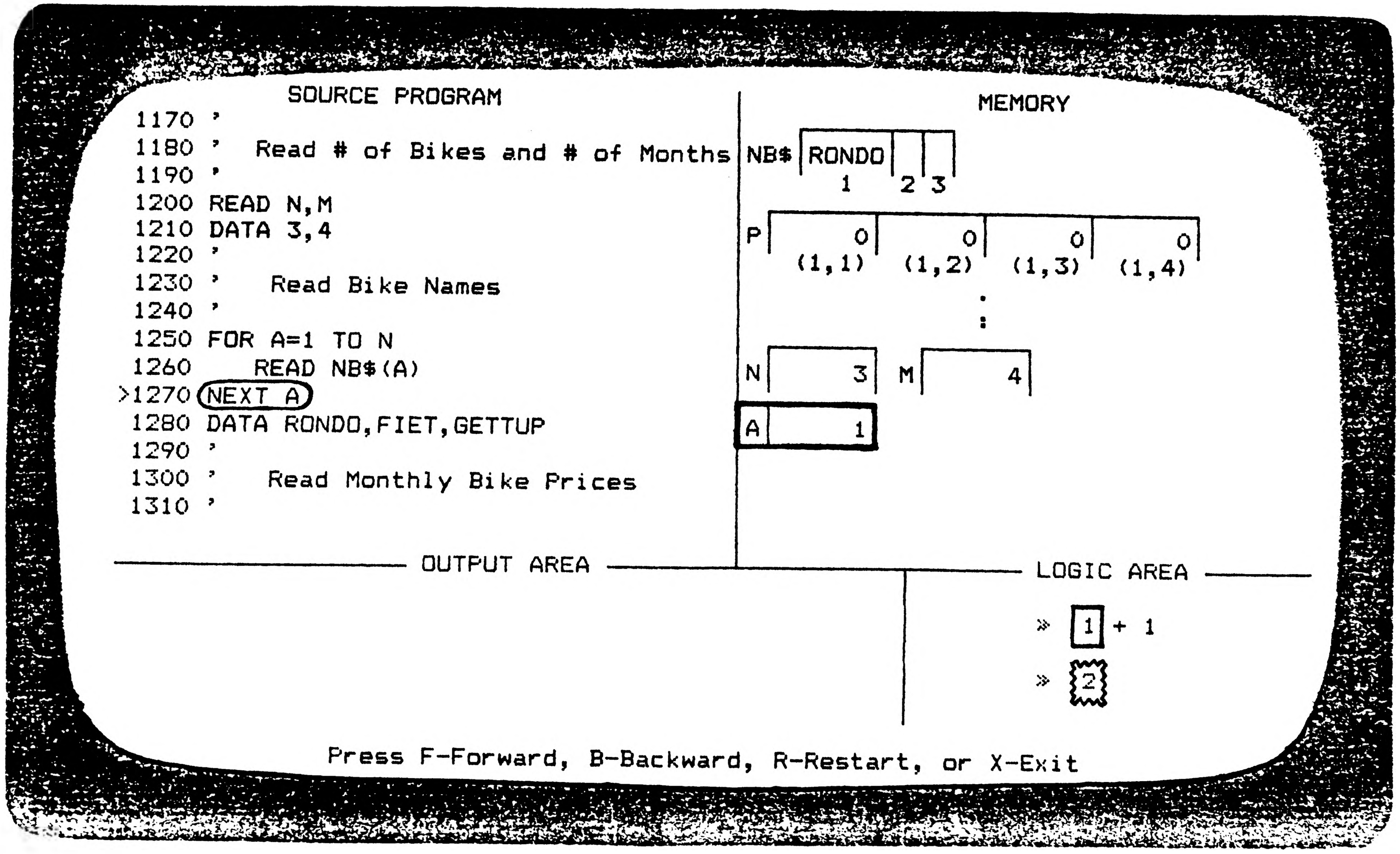
5a. If the incremented value of the index variable has



```
 1170 '
 1180 '  Read # of Bikes and # of Months
 1190 '
 1200 READ N,M
 1210 DATA 3,4
 1220 '
 1230 '   Read Bike Names
 1240 '
>1250 FOR A=1 TO N
 1260    READ NB$(A)
 1270 NEXT A
 1280 DATA RONDO,FIET,GETTUP
 1290 '
 1300 '   Read Monthly Bike Prices
 1310 '
```

Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 3. Initialization of a FOR-NEXT Loop Index Variable



SOURCE PROGRAM

```
 1170 '
 1180 '   Read # of Bikes and # of Months
 1190 '
 1200 READ N,M
 1210 DATA 3,4
 1220 '
 1230 '    Read Bike Names
 1240 '
 1250 FOR A=1 TO N
 1260     READ NB$(A)
>1270 NEXT A
 1280 DATA RONDO,FIET,GETTUP
 1290 '
 1300 '    Read Monthly Bike Prices
 1310 '
```

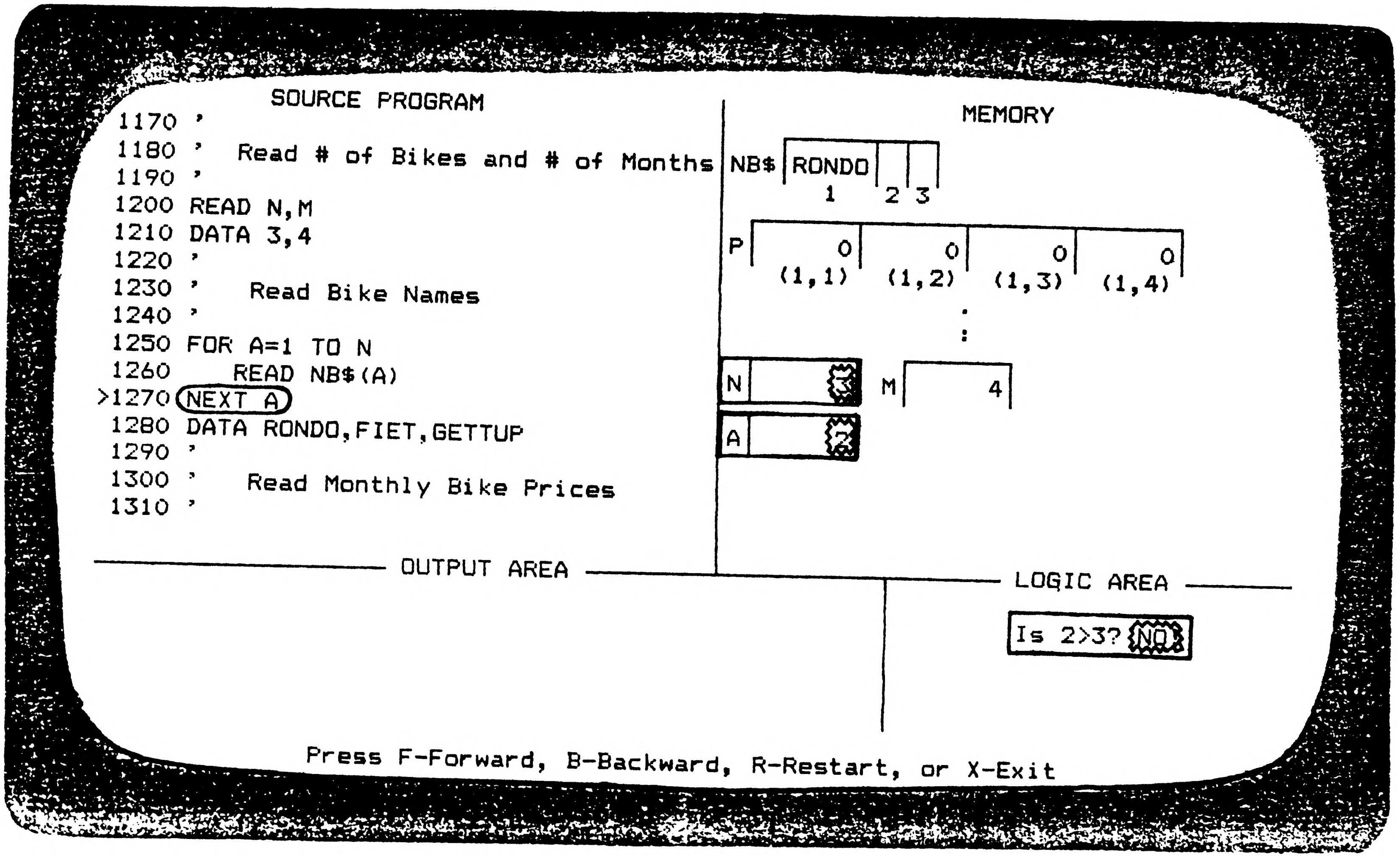
Figure 4. Incrementation of a FOR-NEXT Loop Index Variable

not exceeded the terminal value then the body of the loop is processed another time. Figure 5 is an example of this step.

5b. If the incremented value of the index variable is greater than the terminal value the program continues executing with the statement after the NEXT statement. Figure 6 shows this test. Upon completion of the loop, the index variable contains the last value that exceeds the limit.

One of the main reasons for using arrays is to be able to write a loop to perform the same operation on each element of the array. When loop counters are used as subscripts, the loop variable and its value are highlighted in the MEMORY window. This is to aid identifying the correct elements located elsewhere in MEMORY.

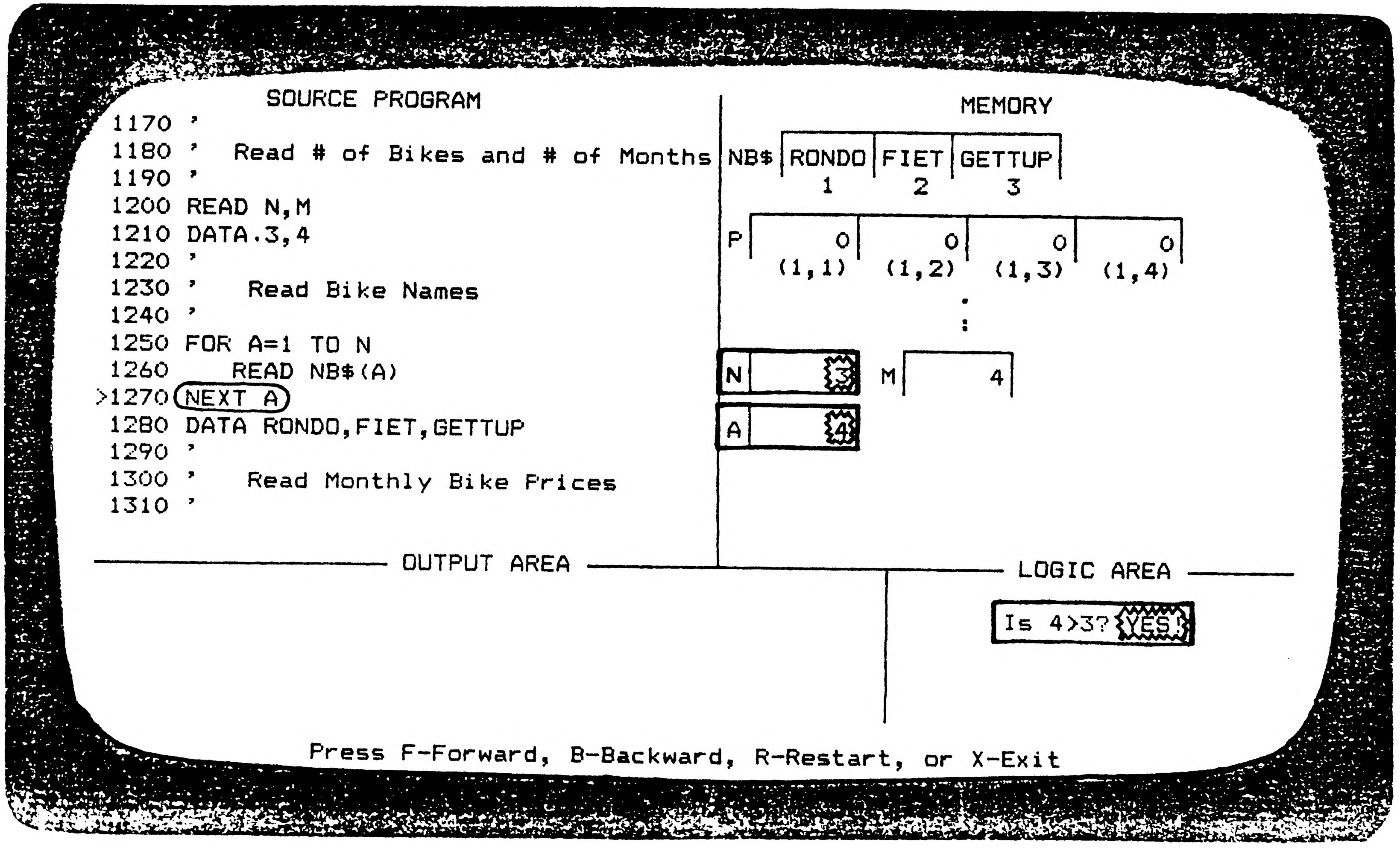
The LOGIC window also shows the computation taking place in an Arithmetic statement. As the statement in the SOURCE PROGRAM is executed, the hierarchical order of operations takes precedence as in the evaluation of an algebraic expression. The two operands and their operator are the only items shown in high-intensity in the SOURCE PROGRAM window. The operator and the values of the operands are depicted in the LOGIC window in reverse video. The result of the operation is shown in reverse video and blinking. The variables and their values are displayed in MEMORY in reverse video. This is depicted in Figure 7. The operator with the next highest precedence and its operands are displayed in the LOGIC window of the subsequent screen. The new operands and their operator are now isolated in the



```
 1170 '
 1180 '  Read # of Bikes and # of Months
 1190 '
 1200 READ N,M
 1210 DATA 3,4
 1220 '
 1230 '   Read Bike Names
 1240 '
 1250 FOR A=1 TO N
 1260    READ NB$(A)
>1270 NEXT A
 1280 DATA RONDO,FIET,GETTUP
 1290 '
 1300 '   Read Monthly Bike Prices
 1310 '
```

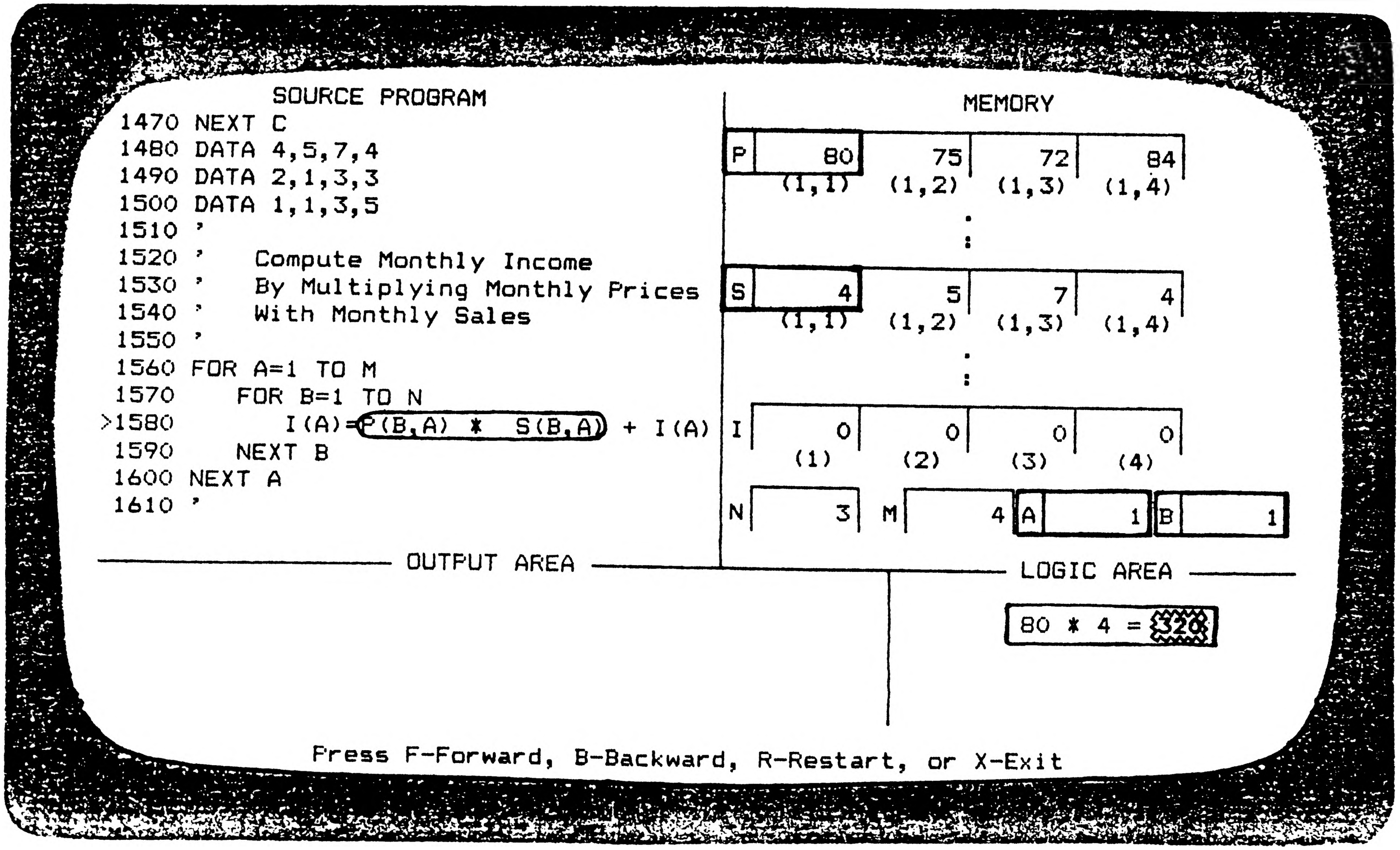
Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 5. Loop Termination Test--Loop to be Repeated


```
                SOURCE PROGRAM
 1170 '
 1180 '   Read # of Bikes and # of Months
 1190 '
 1200 READ N,M
 1210 DATA.3,4
 1220 '
 1230 '    Read Bike Names
 1240 '
 1250 FOR A=1  TO N
 1260     READ NB$(A)
>1270 NEXT A
 1280 DATA RONDO,FIET,GETTUP
 1290 '
 1300 '    Read Monthly Bike Prices
 1310 '
```

Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 6. Loop Termination Test--Loop is Completed



SOURCE PROGRAM

```
 1470 NEXT C
 1480 DATA 4,5,7,4
 1490 DATA 2,1,3,3
 1500 DATA 1,1,3,5
 1510 '
 1520 '   Compute Monthly Income
 1530 '   By Multiplying Monthly Prices
 1540 '   With Monthly Sales
 1550 '
 1560 FOR A=1 TO M
 1570    FOR B=1 TO N
>1580       I(A)=P(B,A) *  S(B,A) + I(A)
 1590    NEXT B
 1600 NEXT A
 1610 '
```

MEMORY
P 80 (1,1) 75 (1,2) 72 (1,3) 84 (1,4)
S 4 (1,1) 5 (1,2) 7 (1,3) 4 (1,4)
I 0 (1) 0 (2) 0 (3) 0 (4)
N 3
M 4
A 1
B 1
OUTPUT AREA
LOGIC AREA
80 * 4 = 320

Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 7. First Operation in an Arithmetic Statement

SOURCE PROGRAM window by being displayed in high-intensity. Refer to Figure 8. This procedure is continued until the arithmetic expression is simplified to a single value. This value is then assigned to the variable. This assignment is depicted in MEMORY as usual, but in the SOURCE PROGRAM only the receiving variable and the assignment symbol are highlighted. See Figure 9.

C. OPERATION

The system is designed to emulate a computer with a 'single-instruction-execute' key. Each time the student presses a key to advance to the next statement in the program, the screen changes to reflect approximately one machine instruction being executed. Initially, the student is started at the first screen and instructed to press a key to proceed forward to the next screen. Continuing this procedure the student could see the execution of the entire program.

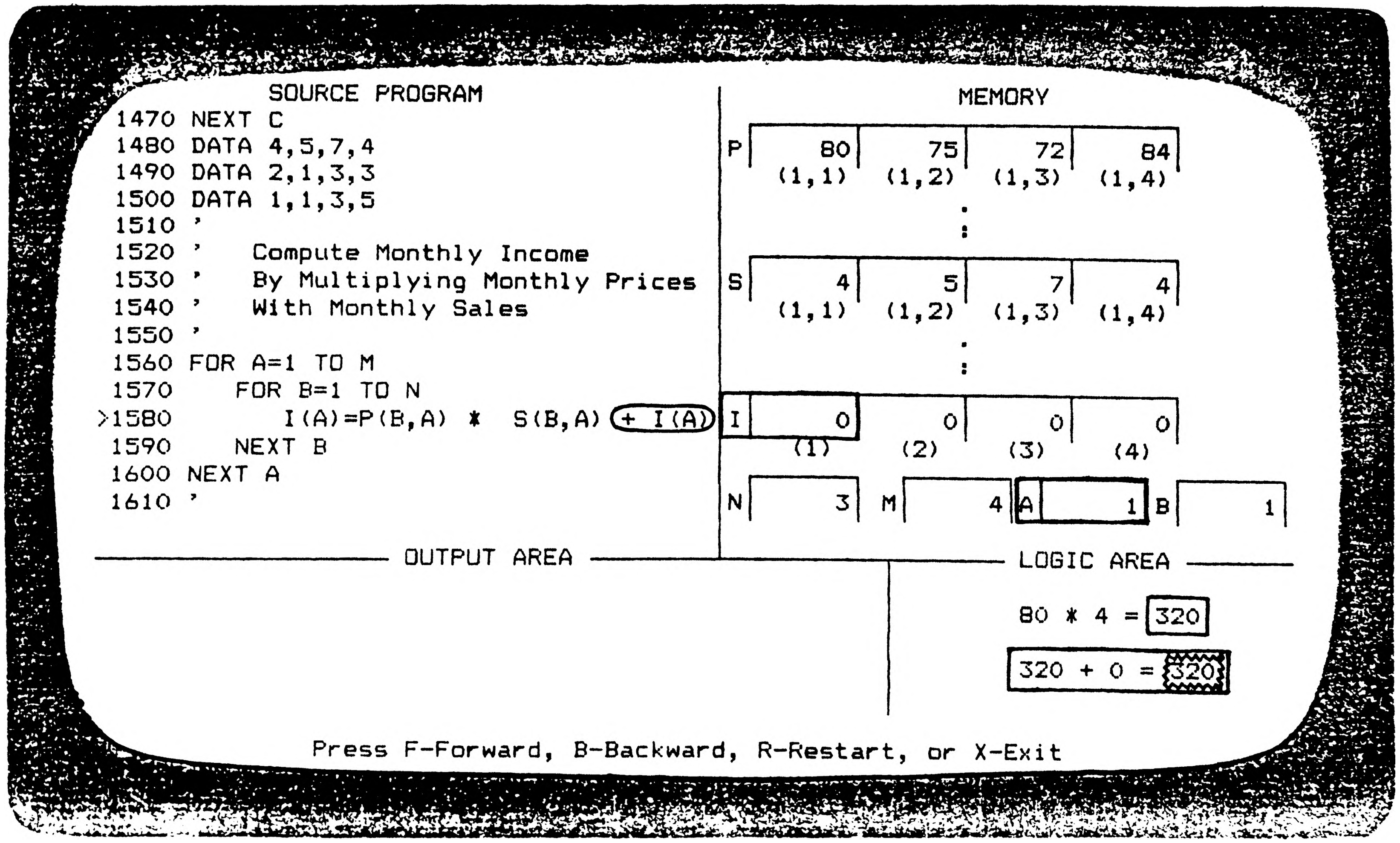
To allow the student the flexibility of reviewing a previous screen, two options are available:

1. The student is able to go back one screen at a time (i.e., 'uncompute'), including all the way back to the beginning; or

2. There is a restart, which will automatically return to the very beginning.

If an attempt is made to go back farther than the beginning, the following message is displayed: "You are at the beginning and may not go back further." The first screen is then displayed for viewing by the student.

An exit from the program is included to complete the



SOURCE PROGRAM

```
 1470 NEXT C
 1480 DATA 4,5,7,4
 1490 DATA 2,1,3,3
 1500 DATA 1,1,3,5
 1510 '
 1520 '   Compute Monthly Income
 1530 '   By Multiplying Monthly Prices
 1540 '   With Monthly Sales
 1550 '
 1560 FOR A=1 TO M
 1570    FOR B=1 TO N
>1580       I(A)=P(B,A) *  S(B,A) + I(A)
 1590    NEXT B
 1600 NEXT A
 1610 '
```

MEMORY

P	80	75	72	84
	(1,1)	(1,2)	(1,3)	(1,4)
S	4	5	7	4
	(1,1)	(1,2)	(1,3)	(1,4)
I	0	0	0	0
	(1)	(2)	(3)	(4)

N	3	M	4	A	1	B	1

OUTPUT AREA

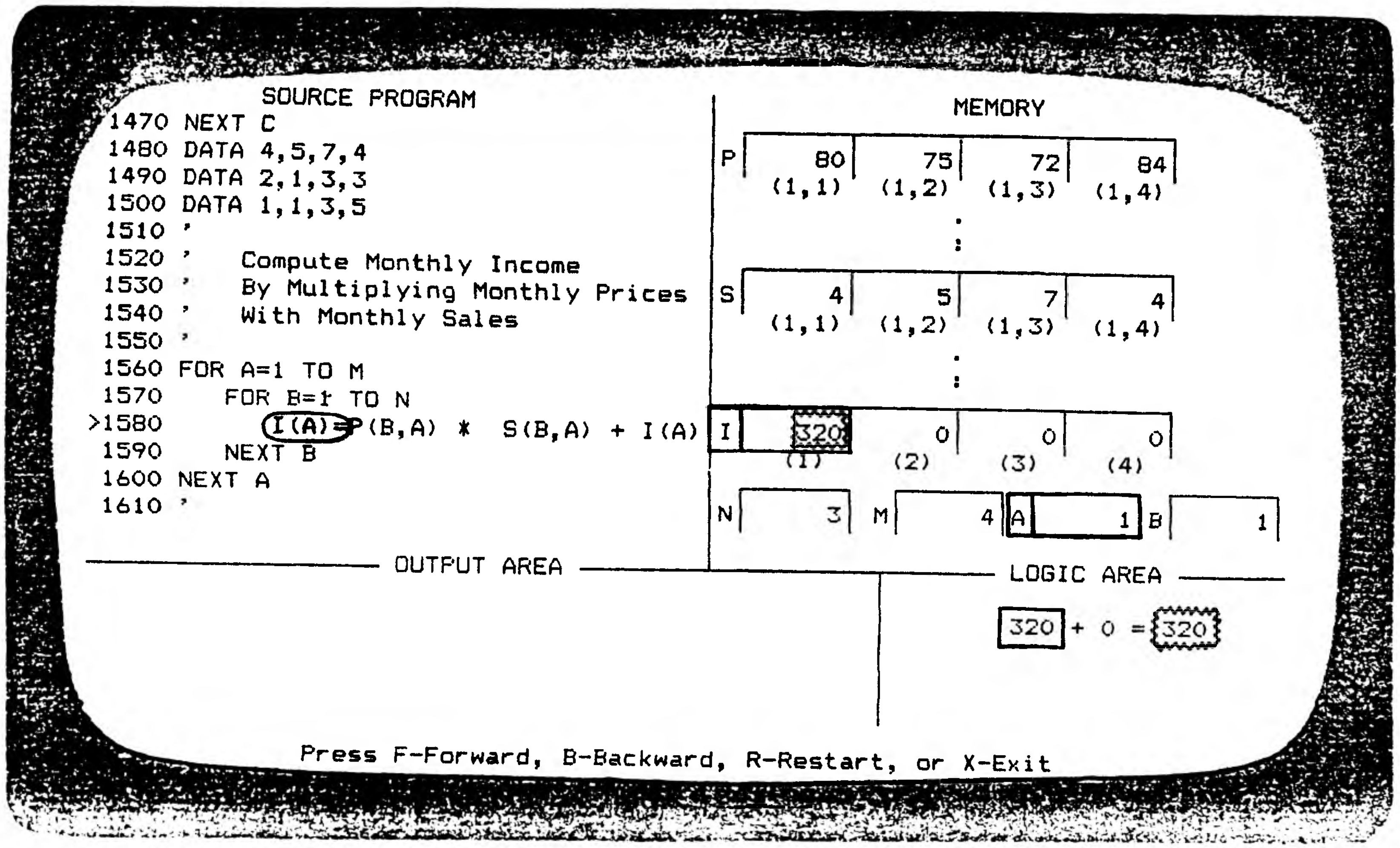
LOGIC AREA

80 * 4 = 320

320 + 0 = 320

Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 8. Second Operation in an Arithmetic Statement



```
 1470 NEXT C
 1480 DATA 4,5,7,4
 1490 DATA 2,1,3,3
 1500 DATA 1,1,3,5
 1510 '
 1520 '   Compute Monthly Income
 1530 '   By Multiplying Monthly Prices
 1540 '   With Monthly Sales
 1550 '
 1560 FOR A=1 TO M
 1570    FOR B=1 TO N
>1580      I(A)=P(B,A) * S(B,A) + I(A)
 1590    NEXT B
 1600 NEXT A
 1610 '
```

```
P     80     75     72     84
   (1,1)  (1,2)  (1,3)  (1,4)
S      4      5      7      4
   (1,1)  (1,2)  (1,3)  (1,4)
I    320      0      0      0
     (1)    (2)    (3)    (4)
N 3   M 4   A 1   B 1
```

320 + 0 = 320

Press F-Forward, B-Backward, R-Restart, or X-Exit

Figure 9. Assignment of an Evaluated Arithmetic Expression

list of viewing options. A simple menu is displayed at the bottom of every screen. It directs the student to press F for Forward, B for Backward, R for Restart, or X for EXit.

During initial testing a lack of flexibility became obvious in being allowed to advance or review only one screen at a time. It is time-consuming and annoying to be forced to manuever at that rate. To alleviate this situation, a jumping option is incorporated. The student must press two keys to activate the jump. The first key is a function key indicating the direction of the jump: F1 for Backward; F2 for Forward. The second key is numeric and indicates the size of the jump. The sequence F1-1 would jump backward four screens. The sequence F2-5 would jump forward twenty screens.

D. EVALUATION

A field study was done with a class of eleven students taking their second course in BASIC. The intervening time period since their first course ranged from one month to one year. During the evaluation, each individual had exclusive use of an IBM PC, the same machines they were using in their present class. (All the students had acquired hands-on experience with some type of microcomputer in their first course.)

The following explanations were given.

1. The program is a teaching aid that graphically depicts a program in execution.

2. The viewing options available--F, B, R, X, and screen-jumping--and how to use them.

Students were provided a handout to accompany the screen presentation. The handout contained the programming assignment and the example program solution. (See Figure 10.) The students were instructed to refer to the program listing on the handout as often as they liked. This was done to offer more flexibility in reviewing the entire program. The handout allowed the students to locate the statement being executed with respect to the total program. It also provided a total picture of the program of which they could see only fifteen lines on the screen. At a glance, the student could identify which portion of code had been executed and/or what portion was yet to be executed.

The students were instructed to signal if they needed assistance. The students were encouraged to register their comments, questions, or points of confusion. These could be in written or oral form.

PROGRAMMING PROBLEM

Rollum Wheels requests a program to compute the income for the first four months of the year. The bicycle model names, monthly prices, and monthly sales records for that four-month period are given below.

Write a program that will read in the data and find the accumulated income for each month.

Model	Prices				Sales			
	Jan	Feb	Mar	Apr	Jan	Feb	Mar	Apr
Rondo	80	75	72	84	4	5	7	4
Fiet	65	61	59	62	2	1	3	3
Gettup	22	22	20	22	1	1	3	5

The following BASIC program is one solution to perform the requested task.

```
1000 '   Rollum Wheels Monthly Income
1010 '
1020 '******* Define the Tables ********
1030 OPTION BASE
1040 DEFINT A-Z
1050 '
1060 '   NB$ -- Name of Bicycles
1070 '   P   -- Monthly Prices
1080 '   S   -- Monthly Sales
1090 '   I   -- Monthly Income
1100 '**********************************
1110 DIM NB$(3), P(3,4), S(3,4), I(4)
1120 '
1130 '****** Define the Variables ******
1140 '   N -- Number of Bicycles
1150 '   M -- Number of Months
1160 '**********************************
1170 '
1180 '   Read # of Bikes and # of Months
1190 '
1200 READ N,M
1210 DATA 3,4
1220 '
1230 '   Read Bike Names
1240 '
1250 FOR A=1 TO N
1260    READ NB$(A)
1270 NEXT A
1280 DATA RONDO,FIET,GETTUP
```

Figure 10. Programming Handout

```
1290 '
1300 '   Read Monthly Bike Prices
1310 '
1320 FOR B=1 TO N
1330    FOR C=1 TO M
1340       READ P(B,C)
1350    NEXT C
1360 NEXT B
1370 DATA 80,75,72,84
1380 DATA 65,61,59,62
1390 DATA 22,22,20,22
1400 '
1410 '   Read Monthly Bike Sales
1420 '
1430 FOR C=1 TO N
1440    FOR D=1 TO M
1450       READ S(C,D)
1460    NEXT D
1470 NEXT C
1480 DATA 4,5,7,4
1490 DATA 2,1,3,3
1500 DATA 1,1,3,5
1510 '
1520 '   Compute Monthly Income
1530 '   By Multiplying Monthly Prices
1540 '   With Monthly Sales
1550 '
1560 FOR A=1 TO M
1570    FOR B=1 TO N
1580       I(A)=P(B,A) * S(B,A) + I(A)
1590    NEXT B
1600 NEXT A
1610 '
1620 '   Print Headings for
1630 '   Models, Prices, and Sales
1640 '
1650 PRINT TAB(3)"Model";
1660 PRINT TAB(19)"Prices";
1670 PRINT TAB(39)"Sales";
1680 PRINT TAB(14)"JAN FEB MAR APR";
1690 PRINT TAB(34)"JAN FEB MAR APR";
1700 '
1710 '   Print Bicycle Models,
1720 '   Monthly Prices, and Sales
1730 '
1740 FOR A=1 TO 3
1750   PRINT TAB(3)NB$(A);
1760   FOR B=1 TO 4
1770     PRINT TAB(14+(B-1)*4) P(A,B);
1780   NEXT B
1790   FOR C=1 TO 4
1800     PRINT TAB(31+C*4) S(A,C);
```

Figure 10. Programming Handout (continued)

```
1810   NEXT C
1820 NEXT A
1830 PRINT
1840 '
1850 '   Print Headings and Computed
1860 '   Value for Income by Months
1870 '
1880 PRINT
1890 PRINT TAB(29)"Income"
1900 PRINT TAB(23)"JAN  FEB  MAR  APR";
1910 FOR D=1 TO 4
1920   PRINT TAB(18+D*5) I(D);
1930 NEXT D
1940 END
```

The screen presentation will provide a statement-by-statement execution of this program. The format highlights one statement or operation of this program at a time. The screen is divided into four areas: SOURCE PROGRAM, MEMORY, Arithmetic-Logic Unit (labeled LOGIC), and OUTPUT AREA. As the screen progressively highlights a statement or operation, it also illustrates the corresponding action in the appropriate area--MEMORY, LOGIC, or OUTPUT AREA.

Figure 10. Programming Handout (continued)

IV. RESULTS

The results are categorized into two areas: positive comments and suggestions for improvement or enhancement.

A. <u>POSITIVE COMMENTS</u>

1. The program is excellent for showing the locations of data in arrays.
2. The logic section shows the necessary logic very well.
3. The program is helpful for learning the internal activity of memory.
4. It would be a great benefit if individual programs could be run so students could see if their programs executed according to specifications.
5. This would be a good debugging aid.

B. <u>SUGGESTIONS FOR IMPROVEMENT OR ENHANCEMENT</u>

1. The Screen-jumping options should be added to the menu at the bottom of the display screen.
2. The screen rewrite delay causes loss of attention.
3. A HELP function would be beneficial
   a) to provide a narrative description of the topic being covered at that point;
   b) to provide a data dictionary for variable names;
   c) to provide a starting place for a requested topic.
4. Show execution of repetitions rapidly
   i.e., in a loop that READs data for a 3 x 4 array, show the first row being assigned and then show the other two rows in one step each.
5. Display total output at the end of the program on a full screen.

V. DISCUSSION

The positive comments reflect the rationale and design goals for the program. The purpose of the program is to aid learning by showing the workings of a program in execution. This project is a prototype; its general design is to show any program in execution. If it were implemented to execute any program, it could be classified as a debugger.

As in making any prototype into a viable system, there is much room for improvement and enhancement. Each suggestion offered in the RESULTS section is considered with other ideas for enhancement.

Including the screen-jumping option on the screen's menu is a valid request. It should be incorporated into any subsequent effort. Students appear more comfortable with their options clearly defined and displayed.

The delay caused by writing to the screen was anticipated. Technological advances will improve this situation as RAM memory is increased. A flexible look-ahead capability in the reading relationship between video RAM and RAM would be an excellent improvement and would provide the remedy for this situation.

A HELP function, as mentioned in the suggestions, could cover many aspects. The HELP function could provide a narrative description of the relevant topic as suggested. This capability could be enlarged to serve as an on-line programming manual. Also, it could supply system documentation. Documentation for the system and the operating instructions could be readily accessible. As part of the operating instructions, a table of topics could be referenced. This could provide the capability to jump to the start of any topic. The documentation for the program being displayed could include a data dictionary, but this would require interactive documentation.

The repetitious nature of displaying loops was noted during testing. This was the motivation for instituting the screen-jumping option. The capability to jump screens allows the student to either view the details or skip screens once the concept is familiar. When a student becomes comfortable with the operating procedures of the system and acquires confidence in the material covered, the screen-jumping option is more likely to be exercised. (The subject who made this comment had viewed every screen with the fear of missing a hidden quiz.)

It was a most reasonable request to desire to see the entire output after completing the program's execution. This fifth suggestion was not anticipated but could be incorporated easily.

Two enhancements to the system that would facilitate several of these suggestions involve window size and independent window scrolling. These capabilities would allow an expanded window to scroll independently and, thereby, cover more material. For example, the OUTPUT AREA could be enlarged to display all the output. If needed, the window could expand to use the entire screen. A window could be substituted to display a HELP session adjoining the

SOURCE PROGRAM. This would enable the student to check syntax in the source code while referencing the on-line programming manual.

It is noted here that the pointer is incorrectly stationed at the NEXT statement of the loop when the index variable is incremented and is tested against the terminal value. In future versions, a blank line would be inserted after the FOR statement. After the index variable is initialized, the pointer would be positioned at the blank line. Simultaneously, the index variable and the terminal value would be compared in the LOGIC window. Then, each line of the body of the loop would be evaluated as the prototype correctly demonstrates. After the body of the loop is executed, the pointer would return to the blank line located after the FOR statement. Here the index variable would be incremented and tested against the terminal value. This would represent more accurately the correct placement of the increment, test, and the conditional branch in the execution of a loop.

VI. CONCLUSION

With the number of individuals who are learning programming growing rapidly and considering the difficulty involved in learning programming concepts, there is a clear need for systems which help the student at the early stage of learning. "A good programming environment...should be of assistance to the programmer." [WERT81] These systems should be available at the critical time when the student is alone and asks, "Why doesn't my program work?"

By seeing a visual representation a student is able to construct his own mental image. This mental image becomes incorporated into a wide-ranging network of associations. These associations can be connected in many ways to other concepts. Because a box with a letter or letters on it is not new, it can be assimilated into the mental network structure more readily than something new. The new concepts are built upon older, more familiar concepts. Using this simplified technique learning is transferred.

Visual illustrations are more effective than reading or hearing. Illustrations have been helping to convey information and meaning in many areas. Visual aids have become a part of all of education. They are commonly used, in the form of maps and charts, in business meetings to all levels of an organization.

Familiarity breeds success. If a student is comfortable with a concept he is more likely to use it and use it well. By seeing new concepts presented with familiar objects the building blocks of learning are stacked in the students favor.

A sample size of eleven was not adequate to verify statistically the hypothesis. However, the resulting comments were encouraging. The responses indicate that this system would be helpful to the student and should be tried.

BIBLIOGRAPHY

ASHB73	Ashby, G., L. Salmonson, & R. Heilman, "Design of an Interactive Debugger of FORTRAN: MANTIS," Software-Practice and Experience, 3(1973), 65-74.
CONK83	Conklin, Dick, PC Graphics: Charts, Graphs, Games, and Art on the IBM PC, John Wiley and Sons, Inc., New York, 1983.
DAVI82	Davies, J.J., "Linking Computer Technology and Learning: The Case for Human Teachers and Computer Learners," Educational Technology, 22(10), (October, 1982), 13-17.
ELLI82	Elliott, Brig, "A High-level Debugger for PL/I, Fortran and Basic," Software-Practice and Experience, 12(1982), 331-340.
HART79	Hart, Jolene J., "The Advanced Interactive Debugging System (AIDS)," SIGPLAN Notices, 14(12) December, 1979, 110-121.
HAWK78	Hawkins, D., "Critical Barriers to Science Learning," Outlook, Issue No. 29 (1978), Mountain View Center for Environmental Education, University of Colorado, Boulder.
JOHN79	Johnson, Mark Scott, "Translator Design to Support Run-time Debugging," Software-Practice and Experience, 9(1979), 1035-1041.
LAY81	Lay, R.W., "Basic Techniques for Teaching 'BASIC'," Proceedings of the IFIP TC-3 3rd World Conference of Computers in Education, (1981), 39-42.
MCCL80	McCloskey, M., A. Carramazza, & B. Green, "Curvilinear Motion in the Absence of External Forces: Naive Beliefs About the Motion of Objects", Science, 210(5), (1980), 1139-41.
MILL81	Miller, L.A., "Natural Language Programming: Styles, Strategies, and Contrasts," IBM Systems Journal, 20(2), (1981), 184-215.
MILL79	Miller, M.L., "A Structured Planning and Debugging Environment for Elementary Programming," International Journal of Man-Machine Studies, 11 (1979), 79-95.

SOLO83	Soloway, E., E. Rubin, B. Woolf, J. Bonar, & W. Johnson, "MENO-II: An AI-Based Programming Tutor," Journal of Computer Based Instruction, 10(1 & 2), (Summer 1983), 20-34.
TROW81a	Trowbridge, D.E., & A. Bork, "Computer Based Learning Modules for Early Adolescence," Monitor, (November, 1981), 19-21.
TROW81b	Trowbridge, D.E., & A. Bork, "A Computer Based Dialog for Developing Mathematical Reasoning of Young Adolescents," Proceedings of the National Educational Computing Conference, Denton, Texas, (June, 1981).
TROW81c	Trowbridge, D.E. & L.C. McDermott, "Investigation of Student Understanding of the Concept of Acceleration in One Dimension," American Journal of Physics, 49(3), (1981), 242-53.
TROW80	Trowbridge, D.E., & L.C. McDermott, "Investigation of Student Understanding of the Concept of Velocity in One Dimension," American Journal of Physics, 48(12), (1980), 1020-28.
TULL81	Tullis, T.S., "An Evaluation of Alphanumeric, Graphic, and Color Information Displays," Human Factors, 23, (1981), 541-550.
WERT81	Wertz, Harald, "Some Ideas on the Educational Use of Computers," '81 Proceedings of the Annual Conference, ACM, New York, (November 9-11, 1981), 101-107.
WEST77	Westcourt, K.T., J. Beard, L. Gould, and A. Barr, "Knowledge-based CAI: CINS for individualized curriculum sequencing" (Technical Report 290). Stanford: Institute for Mathematical Studies in the Social Sciences, 1977.

VITA

Sherry Ann Lile was born on October 29, 1953 in Brookfield, Missouri. She received her primary and secondary education from Ethel Consolidated School and Macon County R-IV where she graduated valadictorian in May, 1971. She attended Northeast Missouri State University. While there, she was president of Statalcalgeo, the mathematics club; vice president of Kappa Mu Epsilon, honorary mathematics fraternity; secretary, vice president, and Outstanding Senior of Cardinal Key, honorary sorority; and vice president and Outstanding Member of Alpha Phi Sigma, honorary scholastic fraternity. She was the school's first programming intern; serving with the V.A. in Washington, D.C. in 1974. In 1975 she received a B.S. in Mathematics and B.S.E. in Mathematics-Secondary Education.

She taught high school mathematics classes in St. Louis County for three years. The next three years she was employed as a programmer at McDonnell-Douglas Automation Company. In 1981, she programmed for Empire District Electric Company in Joplin, Mo. From there she accepted an invitation to return to her alma mater and teach computer science and mathematics classes. She was on leave the 1983-84 academic year to complete an M.S. in Computer Science at the University of Missouri-Rolla.