

UNCLASSIFIED

AD NUMBER

ADB015698

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies only; Test and Evaluation; 19 JUL 1974. Other requests shall be referred to Air Force Avionics Laboratory, Wright-Patteson AFB, OH 45433.

AUTHORITY

AFAL ltr, 24 Feb 1978

THIS PAGE IS UNCLASSIFIED

THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

AFAL-TR-73-40

STRUCTURED SOFTWARE STUDY

J. David McGonagle
James P. Anderson & Co.

405 358

TECHNICAL REPORT AFAL-TR-73-40

October 1974



D P
DEC 17 1976
47 A

Distribution limited to U.S. Government agencies only,
Test and Evaluation, 19 July 1974.

Other requests for this document must be referred to the
Air Force Avionics Laboratory AAT , Wright-Patterson AFB, Ohio 45433

Air Force Avionics Laboratory
Air Force Systems Command
Wright-Patterson Air Force Base, Ohio

DDC FILE COPY

ADB015698

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

STRUCTURED SOFTWARE STUDY

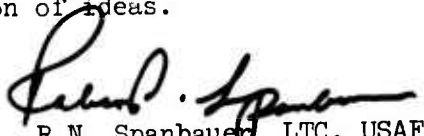
J. David McGonagle

410	
870	
10	
20	
30	
40	
50	
60	
70	
80	
90	
100	
110	
120	
130	
140	
150	
160	
170	
180	
190	
200	
210	
220	
230	
240	
250	
260	
270	
280	
290	
300	
310	
320	
330	
340	
350	
360	
370	
380	
390	
400	
410	
420	
430	
440	
450	
460	
470	
480	
490	
500	
510	
520	
530	
540	
550	
560	
570	
580	
590	
600	
610	
620	
630	
640	
650	
660	
670	
680	
690	
700	
710	
720	
730	
740	
750	
760	
770	
780	
790	
800	
810	
820	
830	
840	
850	
860	
870	
880	
890	
900	
910	
920	
930	
940	
950	
960	
970	
980	
990	
1000	

FOREWARD

3-4 → This study effort was carried out under Project 6090, Task 01, Work Unit 07 by the James P. Anderson Company, Box 42, Fort Washington, Pennsylvania 19034. The principal investigator was J. David McGonagle of the J.D. McGonagle Company, 246 Howarth Road, Media, Pennsylvania 19063. Paul G. Stokholm, Capt. USAF AFAL/AAM was the program monitor. The study was conducted during the period from 1 May 1972 through February 1974. This report was submitted on 29 April 1974.

Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.


R.N. Spanbauer, LTC, USAF
Chief
System Avionics Division

ABSTRACT

This study reports an evaluation of Structured Programming as an aid in the production of highly reliable computer programs. This approach to problem analysis and program composition organizes the program text to clearly reflect the order of execution for the program. The resultant program text reflects the subdivision of the problem into smaller tasks which are clearly identifiable. The rules for performance and ordering of these sub-tasks are reflected in the limited but sufficient set of controls used in program construction. A set of Principles of Structured Programming is developed together with guides for determining an optimal upper and lower bounds on program size. The applications of the Principles to a program are illustrated in the study report. A set of observations and conclusions drawn from the experience of developing a program in this way are presented.

Table of Contents

	<u>Page</u>
I. INTRODUCTION	1
II. A REVIEW OF STRUCTURED PROGRAMMING	3
1. Introduction	3
2. Historical Background	4
3. Basic Concepts	5
4. Principles of Structured Programming	18
III. APPLICATION OF THE PRINCIPLES OF STRUCTURED PROGRAMMING	34
1. Introduction	34
2. Test Problem	34
3. The Program	35
4. Sample Programs as an Illustration of the Principles	50
5. Analyzing Programs for their Correctness	56
IV. THE DIJKSTRA PROGRAMS AND THIS STUDY	66
V. RESULTS AND OBSERVATIONS	75
1. Observation	75
2. Families of Programs	76
3. Program Critical Paths	76
4. Recognizing Common Functions	76
5. Error Detection in Programs	77
6. Memory Efficiency	77
7. Hardware-Software Trade-Offs	78
REFERENCES	79
BIBLIOGRAPHY	81

List of Figures

	<u>Page</u>
Figure 2. 1. Simple Problem Decomposition	6
Figure 2. 2. Second Order Problem Decomposition	7
Figure 2. 3. Two Level Program Text	9
Figure 2. 4. Trace of Control for Execution of PROGRAM1	10
Figure 2. 5. Third Order Problem Decomposition	12
Figure 2. 6. LEVEL3 Programs - No Factoring	13
Figure 2. 7. LEVEL3 and LEVEL4 - Showing Factoring	14
Figure 2. 8. A First Program for Simulations	20
Figure 2. 9. A Program to Perform a Simulation	21
Figure 2.10. Text of PDP11SIM	22
Figure 2.11. Fifth Order Decomposition of Problem	23
Figure 2.12. Level Designations in the Program Text.	25
Figure 2.13. Flow Diagram for the Program PDP11SIM	29
Figure 2.14. Flow Diagram for the Program RUNSIM	31
Figure 2.15. Text of RUNSIM	32
Figure 3. 1. Text of PDP11SIM	36
Figure 3. 2. Flow Diagram for the Program PDP11SIM	37
Figure 3. 3. Text of RUNSIM	39
Figure 3. 4. Flow Diagram for the Program RUNSIM	40
Figure 3. 5. Text of INSTRUCTNHNDLR	42
Figure 3. 6. Flow Diagram for the Program INSTRUCTNHNDLR	43
Figure 3. 7. Text of FINDFORMAT	44
Figure 3. 8. Flow Diagram for the Program FINDFORMAT	45
Figure 3. 9. Text of IEXECUTE	46
Figure 3.10. Flow Diagram for the Program IEXECUTE	47
Figure 3.11. Text of FINDONEOP	48
Figure 3.12. Flow Diagram for the Program FINDONEOP	49

List of Figures (continued)

	<u>Page</u>
Figure 3.13. Text of OPNONE	51
Figure 3.14. Flow Diagram for the Program OPNONE	52
Figure 4. 1. Dijkstra's Prime Number	69

List of Tables

	<u>Page</u>
Table 3. 1. Opfield Contents	61
Table 3. 2. J Assignments All Possible Instruction Patterns	62
Table 3. 3. Source Field as Operation Indicator	64
Table 3. 4. Possible Instruction Bit Patterns at OPNONE (J = 0) . . .	65
Table 3. 5. Bit Patterns Selected at Each Level	65

SECTION I

INTRODUCTION

This report presents the results of a test of the use of Structured Programming as described by Professor Dijkstra. The study has two objectives: 1) determine the suitability of the Dijkstra paper, "Notes on Structured Programming", as a basis for the practice of Structured Programming, and 2) evaluate the applicability of Structured Programming to the generation of aerospace software.

The study is motivated by a recognition of the requirement for improvements in the software generation process. This requirement is based on the high cost associated with the acquisition of the highly reliable software necessary to the aerospace mission. Increases in both the size and the complexity of systems indicate high costs.

The approach to software composition, evaluated in this study, emphasizes steps that lead to a high level of confidence in the correctness of computer programs. An understandable program text is identified as one of the principal vehicles for achieving correct programs. In particular, a solution to a programming task is specified with a program text that organizes the problem into sets of smaller tasks. This specification involves the use of a restricted set of control structures to organize the tasks. It is intended that the resultant text be the principal documentation for the program, and that the text be sufficiently clear to support a convincing demonstration of the program's validity.

This report presents a review of Structured Programming in Section II. The review includes an introduction to the background of the concept of Structured Programming, and the development and illustration of a set of principles of Structured Programming. The necessity for the identification and enumeration of the principles was motivated by failures of early attempts to apply Structured Programming based only on the Dijkstra paper.

Section III presents an orderly development of a segment of a program to illustrate the repeated application of the principles. The application of the principles is discussed. Following this discussion, arguments for the validity of the work are formulated.

In Section IV the programs developed by Dijkstra are discussed. A set of axioms that provide a rationale for Dijkstra's approach are presented. The program form adopted for the study is then compared with the form used by Dijkstra.

Section V of this report presents a set of observations that resulted from the effort. An extensive bibliography of the literature of Structured Programming is included in this report.

Both of the objectives of the study have been addressed by the report. The conclusions of the study are that Structured Programming can help improve the effectiveness of aerospace systems, but that the Dijkstra paper is not sufficient for use as the basis for the Structured Programming approach to software production. Although the principles derived from the Dijkstra paper and illustrated in this report will probably be extended on the basis of experience, they provide sufficient guidance for the development of well structured programs.

SECTION II

A REVIEW OF STRUCTURED PROGRAMMING

1. INTRODUCTION

The term Structured Programming identifies a particular philosophy of programming. A philosophy of programming is a development and application of a body of principles about programming. Programming can be defined as the art or technique of composing algorithms, called programs, for execution by a processor to perform a particular task.

As is implied by its name, Structured Programming is concerned with the organization or structure of programs. In particular, Structured Programming is concerned with the listing or text of the program.

This concern with the program text is based upon its usefulness as the primary record of the program. The text provides a real basis for relating the execution of the program to the problem it is to solve. Therefore, it is important that the text of the program be understandable. A program text is understandable if a person, unfamiliar with the program, can easily read the code to determine what it does and how it operates. If a program text is understandable, then it is possible to establish a clear, definite, and consistent relationship between the problem and the program execution based on the text.

Each of the principles of Structured Programming presented in the latter part of this section has as a goal the formation of an understandable text. The primary goal of Structured Programming, then, is the production of correct and understandable programs, the text of which can adequately support a convincing demonstration of the correctness of the program with respect to the problem.

Secondary goals of Structured Programming involve the generalization and adaptability of the program. To accomplish these aims, the programs are constructed so as to localize the handling of individual parts of the problem to specific, easily identified parts of the program text. These adaptations may be motivated by an anticipation of changes and modifications in the program specifications, to adjust to new requirements. Alterations in the program may also be motivated by efficiency considerations. These changes are easier if their effects are localized in the program.

Advocates of Structured Programming base their approach to program construction upon the capability of a programmer to arrange or structure the program text according to any pre-determined criteria. The text can be arranged or structured to provide the desired simple and direct relationship between itself, the static form,

and the dynamic or executorial form of the program. The text can then be made the basis for assertions about the properties of, and the validity of the dynamic form of the program. The size and the complexity of the program and its sub-structures can be controlled to conform to the perceptual limitations of the programmer. These disciplined actions aid both understanding and validation.

The concrete realization of all of these basic philosophical beliefs is a usefully structured or a well structured program.

Although the primary proponents of Structured Programming are concerned with achieving a philosophically pure way of deriving a well structured program, there is currently no guaranteed "cookbook recipe" with which to produce one. The primary ingredients, as in all programming, are the ingenuity, experience, and reasoning ability of the programmer.

2. HISTORICAL BACKGROUND

The most prominent names among the proponents of Structured Programming include E. Dijkstra, N. Wirth, B. Randell, C. A. R. Hoare, and H. Mills. Stimulation for the idea of Structured Programming originated from a letter submitted by E. Dijkstra and published under the title, "GOTO Statement Considered Harmful."¹ Most of the organized work in the field traces back to a previously unpublished paper distributed by Dijkstra among members of the computer community.² This paper has recently been published in a collection of papers.³ A theoretical basis for Structured Programming was originally established by Bohm and Jacopini,⁴ and Hoare⁵ and extended by Mills.⁶

Many of the ideas expressed by Dijkstra had previously been incorporated in good programming practice. These ideas were also discussed in the literature in a disjointed intuitive way. However, Dijkstra's paper contains a lengthy philosophical discussion regarding the desirability of organizing the structure inherent in programs in a useful way so as to produce well structured programs. In addition, his paper contains a loosely formulated set of principles to guide the programmer in the construction of such programs, together with some samples of the construction of well structured programs. It is the purpose of these principles to aid the programmer in achieving the specified goals. Program texts organized (structured) according to the principles can be more readily comprehended by others as well as by the original programmer. A determination of the validity of the program, with respect to its specification, can then possibly be made from an inspection of the text alone. The areas of impact for future modifications to the program can be identified by referring to the program text. The balance of this section deals with these principles of good programming practice.

3. BASIC CONCEPTS

While developing the thesis of demonstrably correct programs, Dijkstra's paper observes the difficulty (near impossibility) of producing correct programs. As Dijkstra pointed out, this difficulty is particularly evident where full use is made of the unconstrained capabilities and sequencing control structures available in modern computers and programming languages. Discussions relevant to this problem may be found throughout the literature. The controversy over the GOTO statement is particularly relevant. 1, 7, 8, 9, 10, 11

As an alternative to the undisciplined use of generalized sequencing and control structures a controlled and restricted sequencing discipline, shown by Bohm and Jacopini⁴ to be sufficient for the production of any program, is adopted.

It is proposed that programs be written primarily as an ordered sequence of steps. The statements that make up a program are to be executed in the order in which they are written, with no backward reference.

It is asserted that a program structured this way is the easiest of all program organizations to understand. The text of such a program bears a one-to-one relationship to the execution form of the program. This makes it easier to formulate a convincing demonstration of the correctness of an execution, based upon the text.⁵

It is interesting to note that the basic paper, as well as later papers, stresses a "convincing" demonstration rather than a more formal "proof" of correctness. The notion of being able to demonstrate the correctness of a program stated as a list of an ordered set of steps has strong intuitive appeal. There is a first, second, third, and ultimately an nth step in this program organization. Each step follows its predecessor in a systematic way. The effect of the program's execution can be analyzed through the application of stepwise enumerative reasoning.

This seems to suggest that one has some chance to prove the correctness of such a program. Possibly some of the techniques reported by London,^{12, 13} Good,¹⁴ and others¹⁵ could be useful in constructing such proofs.

From the philosophical point of view, the most primitive operation of Structured Programming is the division of the problem to be solved into two or more smaller problems or subtasks. In the interest of simplicity the subtasks are ordered. This order is expressed by the relative positioning of each of the subtasks in a list.

When these subtasks are carried out or executed, in the order specified, then the effect of their execution is the desired result - a solution to the problem. The sequential ordering of the subtasks accommodates any time dependence that might exist between the subtasks. Figure 2.1 illustrates a decomposition of a simple abstract problem into a sequence of subtasks.

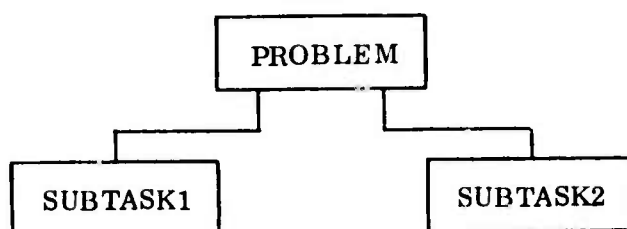


Figure 2.1. Simple Problem Decomposition

When this partitioning of the problem is expressed as a program, the subtasks are carried out by the execution of programs in the order specified, as shown below:

PROGRAM1: begin PE1; PE2; end PROGRAM1;

where PE1 performs SUBTASK1 and PE2 performs SUBTASK2. When the program is executed, the control of the process to solve the original problem rests in PROGRAM1. Whenever PROGRAM1 is executed the execution begins with the first step, which performs SUBTASK1. After PE1 is completed, control is returned to PROGRAM1 for sequencing to the next step. At this point PE2 is executed. When this step is completed, then the execution of PROGRAM1 is completed. The program has been executed in a purely sequential manner with each step carried out exactly once in the order stated. The program consists of two steps or program elements (PE's). A PE can take the form of another program, a subprogram, a macro instruction or a machine instruction, as appropriate.

The orderly translation of the problem decomposition into this program organization establishes the relationship between the problem and the program text. Partitioning the problem and the program along the same boundaries as demonstrated in the above example exposes and preserves the analysis of the problem. The handling of the parts of the problem in this way also localizes them to specific parts of the program text.

The program text, written to express the problem decomposition, is laid out in the order of execution for the program. The simple sequential ordering from top to bottom (beginning to end) satisfies the goals enumerated earlier for Structured Programming (e.g., clear, correct, understandable and adaptable).

If, at this time, the problem has been decomposed into subtasks, all of which are satisfied by PE's which are defined, the decomposition process is completed. A PE is considered to be defined if: 1) there exist programs in a library of programs or instructions in the instruction repertoire which can carry out the subtask and produce

the desired result; or 2) it has occurred previously in the decomposition process. The latter case is considered to be a factorable PE - a program element the execution of which is invoked from more than one place in the program structure.

In most instances the program reflecting the first step of decomposition has some PE's which are undefined, and thus does not meet the criteria for completion.

Whenever any of the PE's are undefined, the task to be performed by that PE can be treated as a problem to be solved. Partitioning is then repeated as often as is required. The result of this partitioning is expressed as a program, just as was done for the original decomposition.

A second application of the decomposition operation creates subtasks of the subtasks, as the partitioning operation is applied to the subtasks derived in the first decomposition. As successive decompositions of the problem are carried out, more of the details involved in the problem and its solution are incorporated. To illustrate this point, a second order decomposition of the abstract problem is presented in Figure 2.2.

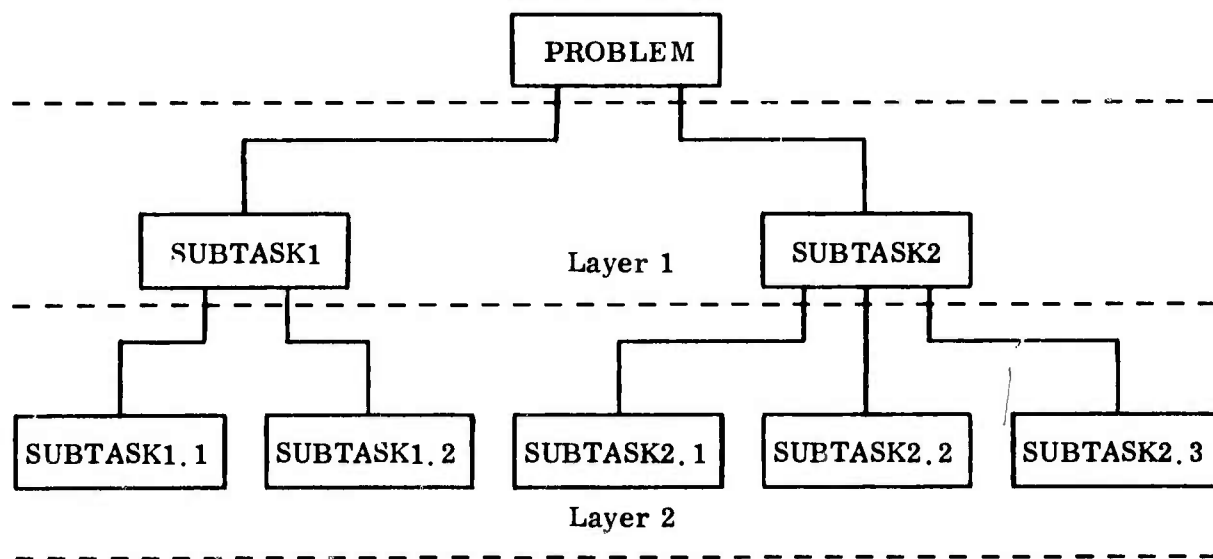


Figure 2.2. Second Order Problem Decomposition

Figure 2.2 illustrates the ordering of the subtasks that result from the second order application of the primitive operation, decomposition. The subtasks shown in the figure are stratified into layers according to the order of the decomposition from which they resulted. Layer 1 contains the set of subtasks which result from the

decomposition of the problem. Layer 2, on the other hand, contains the set of subtasks formed by the decomposition of the members of Layer 1. It is not essential that all members of the layer be decomposed further; some may already be completely defined. This tree-like organization of the partitioned process, in layers, represents one of the possible stratifications of the task. A different decomposition of the problem would yield a different structure.

It is not the hierarchical tree formed by the decomposition of the problem that is of primary interest. Instead, the programs constructed from the problem decomposition are the focal point for consideration. It is often helpful to distinguish between the ideas of problem decomposition and program composition. To help separate these two concepts, this study uses different terms for similar notions that occur in each. The term subtask is used when describing the parts which result from the decomposition of a problem. The net effect of a problem or task is accomplished by performing a proper set of subtasks. A task or a subtask is performed by executing a program, and a program is specified by an ordered set of program elements (PE's). The term layer is used to describe the relative location of subtasks when speaking of the nodes of decomposition. The hierarchical grouping of programs and program elements, on the other hand, are described as belonging to levels of the program organization.

For each layer of the problem decomposition, at least one program or program element, PE, is constructed to identify and order the sequence of the subtasks in that layer. A PE then will most often appear in two ways in the program organization: 1) a PE first occurs as a step in a program (e.g., a procedure call) to define another PE; 2) the PE is itself defined at a lower level (higher numbered).

The program presented earlier corresponds to the decomposition of the abstract problem into the first layer of subtasks. That program, PROGRAM1, is the highest level of the program to solve the abstract problem. The second layer of the decomposition, Figure 2.2, illustrates the need for more than one program to represent the decomposition. At the second layer there are five additional subtasks, two of which result from the partitioning of SUBTASK1, and three as sub-components of SUBTASK2.

The text of the program composed to solve the decomposed problem is shown in Figure 2.3, in which three programs are defined. These programs are identified and separated into two levels. The program (PROGRAM1) which expresses the first decomposition of the problem is placed in the outermost level (LEVEL1). Each of the steps of this program consist of calls on other programs (PE1 and PE2). These program elements are defined, in the example, at the next level (LEVEL2) of the program structure. If any of the PE's used in the programs PE1 and PE2 are defined with a program, those definitions are placed in a still lower level (i.e., 3 or beyond). In each of the programs a simple ordering of the program steps is preserved. Each element of the structure is to be executed once, in the order specified, each time the program (PROGRAM1) is executed. A trace of the flow of control during execution of the two levels of programs is shown in Figure 2.4.

```

LEVEL1:  begin
comment  outermost level;
          PROGRAM1:  begin
comment    performs problem solution;
                PE1;
                comment    performs SUBTASK1 by causing the
                           execution of the program (PE1) defined
                           in LEVEL2;

                PE2;
                comment    performs SUBTASK2 by causing the
                           execution of the program (PE2) defined
                           in LEVEL2;

          end PROGRAM1;
end LEVEL1;

LEVEL2:  begin
comment  specifies programs referred to in LEVEL1;
          PE1:  begin
comment    performs functions identified as SUBTASK1 when
              executed;
              PE3;
              comment    performs SUBTASK1.1;
              PE4;
              comment    performs SUBTASK1.2;

          end PE1;
          PE2:  begin
comment    performs functions identified as SUBTASK2 when
              executed;
              PE5;
              comment    performs SUBTASK2.1;
              PE6;
              comment    performs SUBTASK2.2;
              PE7;
              comment    performs SUBTASK2.3;

          end PE2;
end LEVEL2;

```

Figure 2.3. Two Level Program Text

```
Enter PROGRAM1
  PROGRAM1, Step 1
    Enter PE1
      PE1, Step 1
        Enter PE3
        EXIT PE3
      PE1, Step 2
        Enter PE4
        EXIT PE4
    EXIT PE1
  PROGRAM1, Step 2
    Enter PE2
      PE2, Step 1
        Enter PE5
        EXIT PE5
      PE2, Step 2
        Enter PE6
        EXIT PE6
      PE2, Step 3
        Enter PE7
        EXIT PE7
    EXIT PE2
  EXIT PROGRAM1
```

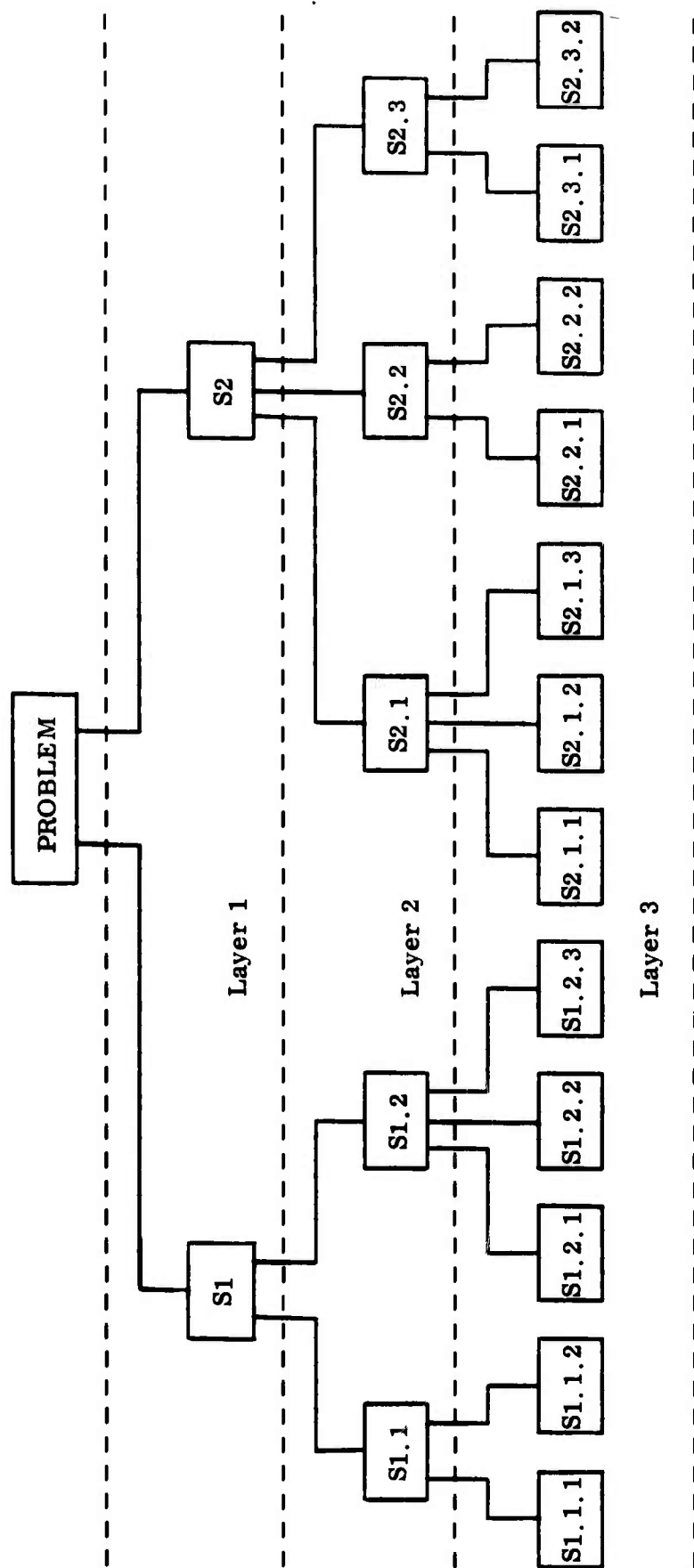
Figure 2.4. Trace of Control for Execution of PROGRAM1

The extremely simple relationships shown in this sample problem are not truly representative of those found in the "real" world. The layers of decomposition in this problem map precisely into the levels of the program organization. This is unusual. No provision is made in the example for the recurrence of the same subtask at different points in the problem. To ignore this is impractical. Common subtasks can often be performed by a single program or program element.

The sample program does show something about the way that PE's are defined, by other PE's at a lower level. The discipline of assigning definitions to lower levels is part of a consistent policy of using only forward references in the program text. The only instance when control is returned to a higher level is when the execution of a program at a lower level is completed. Control then passes back to the calling PE at the higher level. This discipline eliminates a potential cause of obscure circular definitions and dependencies among program elements. Further, it allows testing to be built up from the bottom. The behavior of PE's, then, can be summarized into a set of rules: 1) a PE is activated from and exits to a PE at a higher level (lower numbered); 2) a PE, like all "good" programs, is executed sequentially from its entrance at the top to its exit at the bottom; 3) a PE never communicates directly with another PE at the same level as itself; 4) a PE performs its task by either executing instructions or calls on other PE's, defined at a lower (higher numbered) level.

Having reviewed the rules of discipline for PE's, the problem of handling the recurrence of subtasks can now be addressed. If, during the program decomposition, a subtask recurs (i. e., the decomposition yields a subtask which occurred elsewhere), then the subtask is factorable and the PE which satisfied the earlier occurrence should be used again. If the definition of the PE appears on the same level or a higher level of the program organization, it should be re-assigned to a lower level. The PE definition must be placed at least one level below the last reference to it in the program organization. The re-assignment of the PE implies the re-assignment of any substructure used in its definition.

To illustrate the effect on the program of factoring, it is helpful to develop an additional decomposition of the abstract problem. This decomposition (Figure 2.5) and the resultant LEVEL3 program set (Figure 2.6) are done first without factoring. The program set (LEVEL3) is appended to the previous levels (Figure 2.3). This program set uses a new PE for each subtask. The numbers in the PE labels are used only to indicate uniqueness. An alternate program organization (Figure 2.7) is then developed on the premise that SUBTASK2.2.2 is the same as SUBTASK1.2. When the SUBTASK2.2 is translated into a program element in the LEVEL3 structure, SUBTASK2.2.2 is carried out using PE4. The definition of PE4 which occurs on LEVEL3 in Figure 2.6 is moved to the newly created LEVEL4, in Figure 2.7. LEVEL4 is required to provide a lower level position for the program which defines PE4, since it is called in LEVEL3. From this it can be seen that the program



NOTE: SUBTASK is indicated by an S.

Figure 2.5. Third Order Problem Decomposition

```

LEVEL1: begin
comment      outermost level;
PROGRAM1: begin
comment      performs problem solution;
PE1:
comment      performs SUBTASK1;
PE2:
comment      performs SUBTASK2;
      end PROGRAM1;
end LEVEL1;

LEVEL2: begin
comment      contains programs referred to in LEVEL1;
PE1: begin
comment      performs function identified as SUBTASK1,
              when executed;
PE3:
comment      performs SUBTASK1.1;
PE4:
comment      performs SUBTASK1.2;
      end PE1;
      end PE2;

PE6: begin
comment      performs function identified as SUBTASK2,
              when executed;
PE5:
comment      performs SUBTASK2.1;
PE6:
comment      performs SUBTASK2.2;
PE7:
comment      performs SUBTASK2.3;
      end PE6;
      end PE7;
end LEVEL2;

LEVEL3: begin
comment      this level may contain programs referred to in LEVEL2
              or above. Programs in this level may not refer to other
              programs at this or any higher level;
PE3: begin
comment      performs function identified as SUBTASK1.1;
PE8:
comment      performs SUBTASK1.1.1;
PE9:
comment      performs SUBTASK1.1.2;
      end PE3;
      end PE3;

```

```

PE4: begin
comment      performs function identified as SUBTASK1.2;
PE10:
comment      performs SUBTASK1.2.1;
PE11:
comment      performs SUBTASK1.2.2;
PE12:
comment      performs SUBTASK1.2.3;
      end PE4;

PE5: begin
comment      performs function identified as SUBTASK2.1;
PE13:
comment      performs SUBTASK2.1.1;
PE14:
comment      performs SUBTASK2.1.2;
PE15:
comment      performs SUBTASK2.1.3;
      end PE5;

PE6: begin
comment      performs function identified as SUBTASK2.2;
PE16:
comment      performs SUBTASK2.2.1;
PE17:
comment      performs SUBTASK2.2.2;
      end PE6;

PE7: begin
comment      performs function identified as SUBTASK2.3;
PE18:
comment      performs SUBTASK2.3.1;
PE19:
comment      performs SUBTASK2.3.2;
      end PE7;
end LEVEL3;

```

Figure 2.6. LEVEL 3 Programs - No Factoring

```

LEVEL1: begin
comment    outermost level;
PROGRAM1: begin
comment    performs problem solution;
PE1;
comment    performs SUBTASK1;
PE2;
comment    performs SUBTASK2;
end PROGRAM1;
end LEVEL1;

LEVEL2: begin
comment    contains programs referred to in LEVEL1;
PE1: comment
      performs function identified as SUBTASK1,
      when executed;
PE3;
comment    performs SUBTASK1.1;
PE4;
comment    performs SUBTASK1.2;
end PE1;
PE2: comment
      performs function identified as SUBTASK2,
      when executed;
PE5;
comment    performs SUBTASK2.1;
PE6;
comment    performs SUBTASK2.2;
PE7;
comment    performs SUBTASK2.3;
end PE2;
end LEVEL2;

LEVEL3: begin
comment    this level may contain programs referred to in LEVEL2
           or above. Programs in this level may not refer to other
           programs on this level, or any higher level;
PE3: comment
      performs function identified as SUBTASK1.1;
PE8;
comment    performs SUBTASK1.1.1;
PE9;
comment    performs SUBTASK1.1.2;
end PE3;

PE5: comment
      performs function identified as SUBTASK2.1;
PE13;
comment    performs SUBTASK2.1.1;
PE14;
comment    performs SUBTASK2.1.2;
PE15;
comment    performs SUBTASK2.1.3;
end PE5;

PE6: comment
      performs function identified as SUBTASK2.2;
PE16;
comment    performs SUBTASK2.2.1;
PE4;
comment    performs SUBTASK2.2.2, same as
           SUBTASK1.2;
end PE6;

PE7: comment
      performs function identified as SUBTASK2.3;
PE17;
comment    performs SUBTASK2.3.1;
PE18;
comment    performs SUBTASK2.3.2;
end PE7;
end LEVEL3;

LEVEL4: begin
comment    this level may contain programs referred to in LEVEL3
           or above. Programs in this level may not refer to other
           programs on this level, or any other higher level.
PE4: comment
      performs function identified as SUBTASK1.2
      and SUBTASK2.2.2;
PE10;
comment    performs SUBTASK1.2.1;
PE11;
comment    performs SUBTASK1.2.2;
PE12;
comment    performs SUBTASK1.2.3;
end PE4;
end LEVEL4;

```

Figure 2.7. LEVEL 3 and LEVEL 4 - Showing Factoring

organization, in levels, is not necessarily identical to the problem decomposition of Figure 2.5. A program element (PE) can be referenced anywhere, at any level in the program organization that meets the one restriction: the definition of the PE must occur at a lower level (higher number) than any of its "calls".

The freedom to assign program element definitions to appropriate levels of the program structure can be helpful in increasing the understandability of a program text. Elements which pertain to similar or related aspects of the problem can be collected into a single level as long as they do not refer to each other. Thus the levels are used to further isolate the parts of the problem to specific areas of the program text.

Before presenting the possible variants on the simple ordering control structure that can be used, it is worthwhile to re-state the relationship between a subtask (derived from the decomposition of the problem) and its companion program element. The subtask is a statement of the net effect to be achieved by the execution of the program element. Therefore, the subtask is the criteria for evaluating the appropriateness or correctness of the output of the program element.

All subtasks do not decompose readily into a short, determinant sequence of smaller tasks, as shown in the abstract problem above. Sometimes a problem decomposes into a number of repetitions of the same subtask. An example of this kind of decomposition can be seen at the outermost level of the problem used in this study. When this happens, a repetition control construct can be used to express the decomposition. The net effect of the repetition of a program element can be readily identified (e. g. , determine the square root of a number or run the appropriate number of simulations). This net effect can then be treated as a sequential element in the problem decomposition process. The problem may not have been decomposed in such a way that the effect noted already appears as a subtask.

It was indicated earlier, however, that in the interest of clarity of the program text, the partitioning of the program and the problem is to be made along the same boundaries. Therefore, in this case, it is necessary to re-partition the problem so that the net effect of the program implementation appears as a subtask in the decomposition. Two forms of repetition control that preserve the isolation of the net effect are described later in this section. (Repeat S until Condition; while Condition do S.)

Sometimes a net effect must be achieved by selecting between alternative actions. For example: to find an element having a given key in an ordered list, the average search time can often be shortened by starting in the middle and searching in the appropriate direction. Or, the actions required to form the absolute value of a real number are dependent upon the sign of the number. Thus, under certain conditions of a problem, some of the steps in a program may not be executed. To meet this need there are selection forms of the control statement. The particular forms

of the selection which preserve the net effect (if Condition then S_1 ; if Condition then S_1 else S_2 ; and Case i of $\{S_1, S_2, S_3, \dots, S_n\}$) are described later in this section. A selection statement provides another instance of the interaction between the process of program construction and the problem decomposition process. The net effect of the selection control must appear as a subtask in the problem decomposition. If it does not, then the problem is decomposed again, at the appropriate layer, to yield a subtask which is the net effect of the selection.

Normally the problem decomposition shapes the program organization. However, if actions must be performed which are controlled by a mechanism other than simple sequencing, it may be necessary to alter the program decomposition. The altered decomposition should contain subtasks which match the program organization. One of the subtasks should be the net effect of the program element which uses the non-sequential control mechanism for implementation.

Programs constructed of levels, as illustrated in the abstract problem solution, can be thought of as being a nested set of abstract machines. The machine at LEVEL1 can be thought of as having two (2) instructions, PE1 and PE2. To solve the problem, a program is written for this machine. This program establishes a sequence of the available instructions to achieve a solution to the problem. This program is named PROGRAM1. Other machines can be considered to exist for levels 2, 3 and 4. The machine at LEVEL2 has a five instruction repertoire. The instructions are named: PE3, PE4, PE5, PE6 and PE7. There are two programs currently written for this machine. They are named PE1 and PE2. These programs written for the second level machine provide a meaning to the instructions of the first level machine. When these two abstract machines, each with its own instruction counter and space, are connected, they act as a single machine. When they are connected and a program written for the machine LEVEL1 is executed, both machines are used. The mechanism connecting the various levels of the program can be illustrated by following the execution of the program in this abstract machine organization.

When the program (PROGRAM1) is activated, the instruction pointer for the machine LEVEL1 points to the first instruction of the program. The instruction, PE1, is selected and recognized. Machine LEVEL2 is activated to define the instruction by executing the program PE1. At this time, the instruction pointer of the machine LEVEL2 is set to step 1 of the program PE1, and the instruction PE3 is recognized and executed. After the instruction pointer is stepped, the instruction PE4 is executed. When the instruction pointer is stepped the next time, the end of the program is recognized. The termination of the program in LEVEL2 reactivates the machine LEVEL1. The result of the execution of PE1 is made available to LEVEL1. The machine LEVEL1 increments its instruction pointer to the next instruction. Again, when the instruction is recognized, the next lower level machine, LEVEL2, is activated. This process, in which the two machines work in lock step, is continued

until the higher level machine, LEVEL1, has completed its program. A lower level machine is activated anytime the machine executing the program cannot execute the instruction. Whenever a machine must go to a lower level machine for service, the higher level machine suspends itself and waits until it receives a response.

When program levels are thought of as abstract machines, attention is called to the blend of dependence and independence that exists between the levels. Each of the levels can be thought of as a building block which is itself built of building blocks formed by the lower levels. An approach to the testing of programs is suggested by this building block approach. The lowest level building blocks can be tested and logically validated as independent units. As the lower level units are validated, they are then combined to form the next higher level machine. The programs which are written for each machine define the instruction sets of higher level machines. Thus the validation of the program for any machine confirms the correctness of instructions for a higher (lower number) machine. This approach to testing has been described for an operating system by Dijkstra.¹⁶

An abstract problem has been used as an example in this discussion of some of the basic concepts of Structured Programming. The notions of sequential program construction, building block program organization and top-down program construction (based upon top-down problem decomposition), have all been discussed. Other basic concepts are not easily demonstrated from the abstract model. These concepts relate to the order in which some aspects of program composition should take place and also to the size of the program units developed.

Throughout this discussion, emphasis has been placed on the use of a restricted set of control structures to organize the programs. Experience has shown that a significant number of program errors originate in the control structure. Often the right actions are performed, but the wrong number of times or at the wrong time. This observation has led to a conclusion: the control structure should be isolated from the actions controlled, insofar as possible. In practice this is accomplished partially through the mechanisms of procedure calls, or "program stubs".¹⁷ These stubs indicate the position of an action in the program execution sequence without the necessity to include the details of how it is done. Through the use of this device, the definition of the control structure is given a priority in the order of development over the refinement of the actions being performed.

Deferral of the specification of data representation forms is another of the basic concepts which are helpful in building readable programs. The net effect of the deferral of these program details (actions and data representations) is to isolate the control structure of the program and make it highly visible to the reader of the program text. Isolation of the control from the action appears also to increase both the generality of the program and the evolution of factorable subtasks.

There is no absolute answer to the question of a most appropriate size for program elements. One of the basic premises of the work in Structured Programming has been man's inability to perceive clearly any program that is not small.¹⁸ Various writers have attempted to offer fixed criteria for the size of a program element. The general criteria being proposed is the amount of code that will fit on an 8½ x 11 page, or roughly 50 statements.¹⁹ Possibly this is a reasonable upper limit. It is clear that the degree of perception attained is a function of the complexity of the control structure used in the page. Complex control structures in conjunction with complex arithmetic expressions can easily make even a single page incomprehensible.

The ultimate criteria for the size of a program element must lead to a "completely readable program" text since this is the basis for assertions about the validity of the program. A criteria which relates closely to the content of the page is more appropriate. Perhaps it could be phrased as, "include only as much elaboration as can be grasped and understood in a single reading of the code."

Any aids to readability available through the programming language should be used. Thus comments, indentation of lines and subroutines, together with judicious naming of identifiers, are all incorporated into a well structured program.

It is clear that some languages, such as Algol and PL/1, offer more facilities for improving readability than others. In addition to the appropriate control structures, these languages have an inherent block structure that facilitates the organization of the text to reflect the problem decomposition. The wide support given to PL/1 by IBM makes it reasonable to expect that it will be the dominant language for Structured Programming. The basic concepts of Structured Programming are applicable, however, regardless of the language chosen.

4. PRINCIPLES OF STRUCTURED PROGRAMMING

The basic concepts discussed in the earlier part of this section can be formulated as a set of principles for use in doing Structured Programming. As the practice of Structured Programming spreads it is expected that other principles will be formulated. These principles are a first set and are useful in developing and testing the technique of using Structured Programming. The principles can be applied to the production of correct and understandable code for all problems. The text of those programs developed according to these principles will support either a convincing demonstration of correctness or an analytical detection of their error. The principles are:

- Develop programs from the "top-down", in a way which reflects a top-down decomposition of the problem to be solved.

- Organize the program into distinct levels forming an ordered series of intermediate systems of programs.
- Develop the program as a simple sequence of calls on other programs or statements in a computer language, using only those control structures that preserve the effect of this simple sequence.
- Defer the development of program details until after the control structures are developed.
- Defer decisions about data representation as long as possible.

Each of these principles is discussed in detail below. The examples in the discussion are drawn from the test problem used in the study, the simulation of the DEC PDP-11. *

- a. Principle #1. Develop programs from the "top-down" in a way which reflects a "top-down" decomposition of the problem to be solved.

This principle is the basis for an orderly development of an entire program. It is also the basis for establishing a well-defined relationship between the problem and the execution of the program.

Most problems have a natural focal point, the point the designer considers to be the "heart" of the problem. In the case of the simulation of a computer, the focal point is the processing of an instruction. This is a traditional starting point for the composition of a program to do simulation. It is natural for a programmer to start to outline a program for the test problem as follows:

1. Fetch next instruction;
2. Decode instruction;
3. Execute instruction.

* Digital Equipment Company.

Following this outline, the details about instruction fetching and about each of the other steps are filled in. Attempts are then made to encode the control details required for proper operation, generally working from the inside-out. This natural sequence then starts in the middle of the program and works down and then returns to the middle and works up.

The proper starting point for "top-down" program construction is a general statement of the whole problem. A solution to this problem is specified in a program formed of smaller tasks. These subtasks are themselves then specified in programs. This system of programs reflects a decomposition of the problem, and this process is the fundamental operation of Structured Programming.

Stated simply, the problem used in this study is: "Run as many simulations of the PDP-11 as a user wants." This problem is readily specified as a finite number of executions of a single task. The task being repeated consists of two steps or subtasks. The performance of a single simulation run is the first step in the sequence; the second subtask involves the translation of the user's needs into a control to meet those needs. Assume that there is a program to perform a single simulation run, and identify it as SIMPROG. Further, assume that the user's control is provided by a program, QUERYUSER. When these two program stubs are embedded sequentially into a control structure which provides for the repetition and the proper termination of the sequence, then a complete specification of the problem solution is formed. This specification is given in Figure 2.8.

```

PDP11SIM: begin
boolean   Stop;
repeat    begin
              SIMPROG;
              QUERYUSER (Stop);
            end;
    until Stop;
end PDP11SIM;

```

Figure 2.8. A First Program for Simulations

The program presented is the first program of a "top-down" implementation of the simulator.

Similar "top-down" program construction is applied to each of the small problems identified by the program stubs. As each of the program constructs is

formulated, new, smaller problems are identified. This process is carried on until the problems are specified either by existent programs or by programs composed of executable computer instructions.

The problem represented by the program SIMPROG (inside PDP11SIM) is: "Perform a single simulation of the PDP-11." Any simulation can be partitioned into a sequence of three (3) subtasks:

1. Initialize the simulation;
2. Perform the simulation;
3. Clean up after the simulation.

If each of these tasks were performed by the programs INITSIM, RUNSIM, and CLEANUPSIM, respectively, then a program can be constructed which will specify the meaning of SIMPROG. This program is shown in Figure 2.9.

```
SIMPROG:  begin
          integer    Cyclelimit;
          INITSIM (Cyclelimit);
          RUNSIM (Cyclelimit);
          CLEANUPSIM;
          end SIMPROG;
```

Figure 2.9. A Program to Perform a Simulation

The program presented in Figure 2.9 can be incorporated into the earlier program (Figure 2.8) to form a single program. In the Algol-like language being used for the implementation, this incorporation can be done while still preserving the identity of SIMPROG and the hierarchical ordering between the programs. See Figure 2.10 for the result of the combination.

Starting from the "outermost" statement of the problem, the required controls and the user interfaces are developed as an integrated part of the program. This is in contrast to the "ad hoc" controls and user interfaces that often occur when programs are constructed from other starting points in the problem.

The decomposition of the problem can be presented as a "tree of decomposition." Figure 2.11 presents a five (5) layer decomposition of the problem of the PDP-11 simulation. Each of the layers is formed by the decomposition of a problem specified in the preceding layer.

```

PDP11SIM:  begin
comment   establish iterative control to allow repetition of simulator for
           multiple runs;

```

```

boolean   Stop;

```

```

repeat
  begin
    SIMPROG: begin
    comment   perform a single simulation run;

    integer   Cyclelimit;
    comment   supplied by user to limit the number of
               machine cycles executed;

    INITSIM (Cyclelimit);
    comment   initialize simulator to appropriate I/O and memory
               configuration. Get value for Cyclelimit, let user
               load peripheral files;

    RUNSIM (Cyclelimit);
    comment   perform a run on simulated machine. Use Cyclelimit
               to control against infinite loop in user program;

    CLEANUPSIM;
    comment   cleanup any residual I/O left when RUNSIM quit;

    end SIMPROG;

    QUERYUSER (Stop);
    comment   talk to user and find out if should go again;

    end;
  until Stop;

```

```

comment   end of controlled loop for repeated simulations;

```

```

end PDP11SIM;

```

Figure 2.10. Text of PDP11SIM

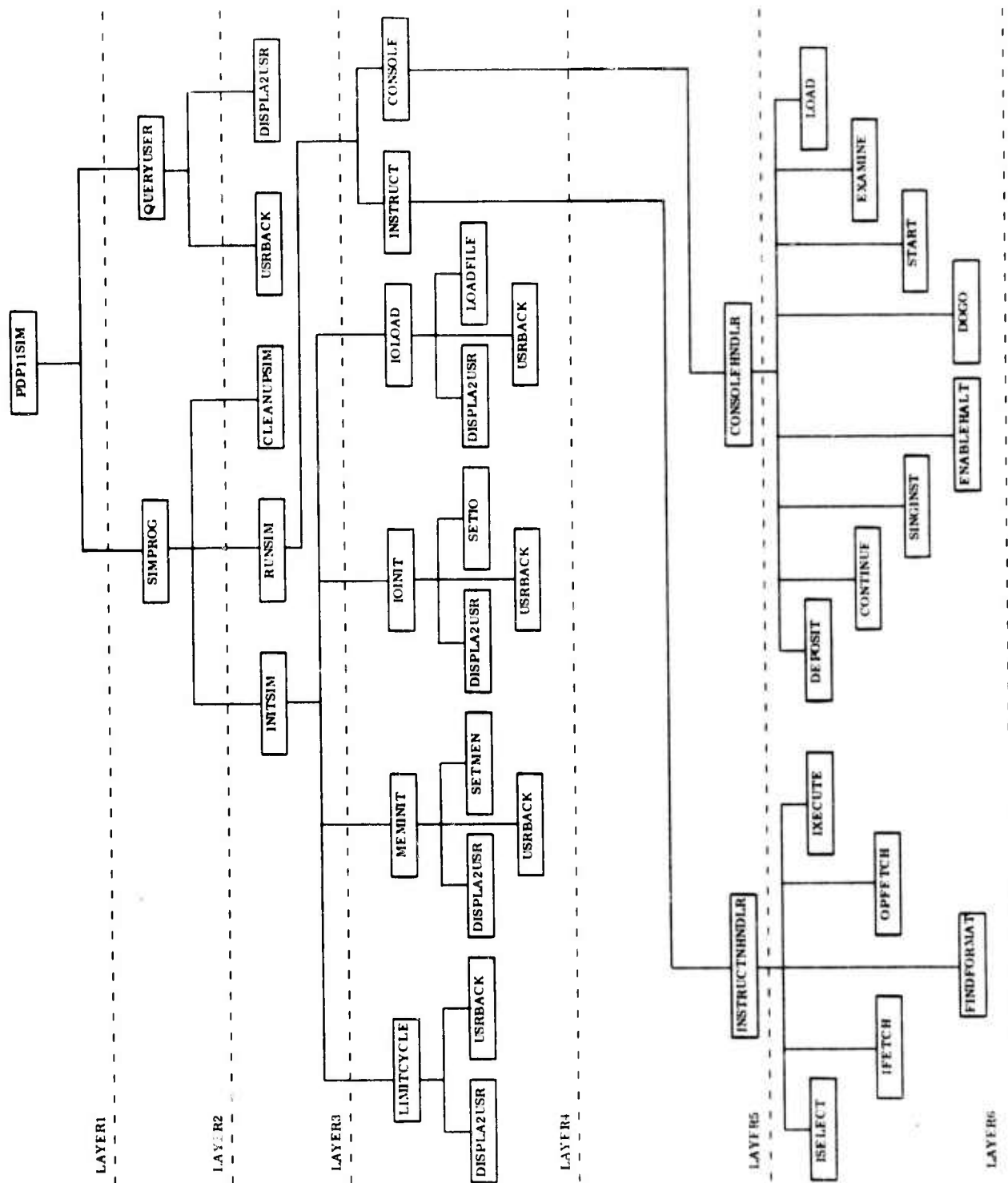


Figure 2.11. Fifth Order Decomposition of Problem

The application of a "top-down" approach to the design of the program does not eliminate the need for re-adjustment to the design. Iterations in the design will occur because alterations to previous work are required. These alterations are caused by decompositions which lead to awkward or incorrect program constructs. These alterations may sometimes require backing up the "tree of decomposition" to higher layers to make changes. The "tree" is often helpful in identifying all of the areas of impact for the alterations. When a new tree and program is built, the decomposition can be continued. The result is usually a program which is a better fit to both the problem and the solution environment. Thus, although the progress toward the ultimate expression of the program is not always uniformly a descending process, it results in a program with a top-down organization.

- b. Principle #2. Organize the program into distinct levels forming an ordered series of intermediate systems of programs.

When a program is composed to reflect a "top-down" decomposition of a problem, it is generally built of a control structure and a series of "calls" or stubs for other programs. For example, the program PDP11SIM (Figure 2.8) contains entry points into two programs, SIMPROG and QUERYUSER. The program SIMPROG (Figure 2.9) is defined by a new program composed of a sequence of three program calls.

This formation of intermediate systems of programs can be repeated to a great depth for a large problem. The arrangement of these programs in the text in an orderly way can present a problem. Obviously, all of them cannot be incorporated into the body of the text if the program is to be readable. In the case of SIMPROG, it was incorporated into the text of PDP11SIM (Figure 2.10) because it was convenient to do so in the Algol-like language. This incorporation is reasonable if: 1) there is no loss of the structural relationships; 2) a simple control structure can be maintained; 3) the generality of the program is preserved; and 4) the "small" size of the program is preserved.

The readability of the program text is improved, however, if these programs are organized into levels which are clearly demarcated in the text. These levels are ordered inversely to their indices, with LEVEL1 the highest level of the program. LEVEL1 always contains one program, which specifies the solution to the general problem statement. In this case, it is PDP11SIM. This program is the entry point for execution of the program, and appears at the start of the text. The levels appear in the text according to their order, together with a notation of any particular significance attached to the level. An outline of the textual organization of the levels in a program text is given in Figure 2.12.

```

LEVEL1:  begin
comment  program at this level is the entry point to controls or specifies
           the total simulation program, providing multiple runs to the user;
           PDP11SIM:  begin
                     .
                     .
                     .
           end PDP11SIM;
end LEVEL1;

LEVEL2:  begin
comment  this level contains the programs to carry out the tasks of the first
           level decomposition of a single simulation problem;
           INITSIM:  begin
                     .
                     .
                     .
           end INITSIM;
           RUNSIM:   begin
                     .
                     .
                     .
           end RUNSIM;
           CLEANUPSIM: begin
                     .
                     .
                     .
           end CLEANUPSIM;
end LEVEL2;

LEVEL3:  begin
         .
         .
         .

end LEVEL3;

```

Figure 2.12. Level Designations in the Program Text

The level to which a program is actually assigned is at the discretion of the programmer. In the example shown in Figure 2.12, LEVEL2 is restricted to programs which deal directly with the performance of a single simulation run. At a lower level in the program organization there is a level dedicated to programs which simulate the effect of PDP-11 machine instructions. Below this is a level which supports these instructions. In this support level, the programs have the effect of performing the logical operations used to define the simulated instructions. This use of the levels in the text to isolate the program elements so as to form levels of abstraction of the problem adds both generality and adaptability to the program.

There is a limitation on the programmer's freedom in the assignment of programs to levels. Common programs, those used in more than one place, appear only once in the program text at a level which is at least one level below the lowest reference to it. In Figure 2.11 there are two (2) subtasks, DISPLA2USR and USRBACK, which appear in several places in the tree. They are elements of QUERYUSER, LIMIT-CYCLE, MEMINIT, IOINIT, and IOLOAD. Based upon the tree as shown, the earliest level at which the programs to perform the subtasks could be placed is below the lowest of these.

The only well-defined criteria for organizing the sub-programs into levels is the matter of interprogram reference (no program on any level may include a reference to a program on the same or a higher level). * The creative use of levels as aids in program organization is very much a function of the individual programmer.

- c. Principle #3. Develop the program as a simple sequence of calls on other programs or statements in a computer language, using only those control structures that preserve the effect of this simple sequence.

In his paper, Dijkstra proposed three kinds of program sequencing constructs (forms) that have the property of providing a beginning-to-end control flow. These are:

- 1) simple sequencing, represented by the concatenation of statements;
- 2) iteration, represented by the forms
while condition do statement
repeat statement until condition;

* For discussion of the reasons for this, see Section II, page

3) selection, represented by the forms

if condition then statement

if condition then statement, else
statement

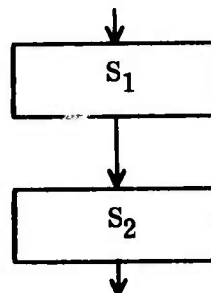
case i of { statement 1, statement 2, ...
statement n }

In these forms, it is important to understand that the statement, S, stands for one statement or a group of statements which, in their effect, act as a single statement. Languages such as Algol 60 and PL/1 use the words begin and end to act as brackets surrounding a group of statements that are treated (in their effect) as a single statement. Such statement groupings are also called compound statements. These sequencing forms have the property of one path in and one path out, as is shown in the following examples given in both flow chart and program form. In the examples, S stands for any statement, C stands for any condition and i is an integer number. Subscripts, if present, designate individual statements.

1) Simple Sequencing

Program form: $S_1; S_2; \dots$

Flow chart:

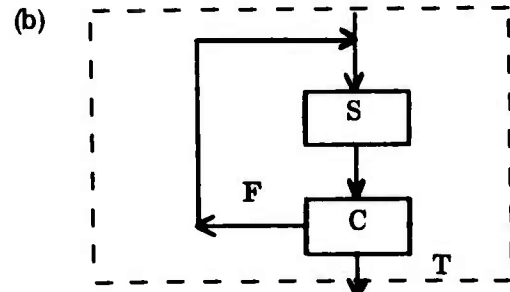
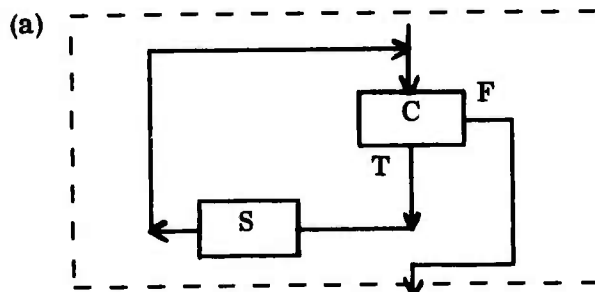


2) Iteration

Program forms: (a) while C do S;

(b) repeat S until C;

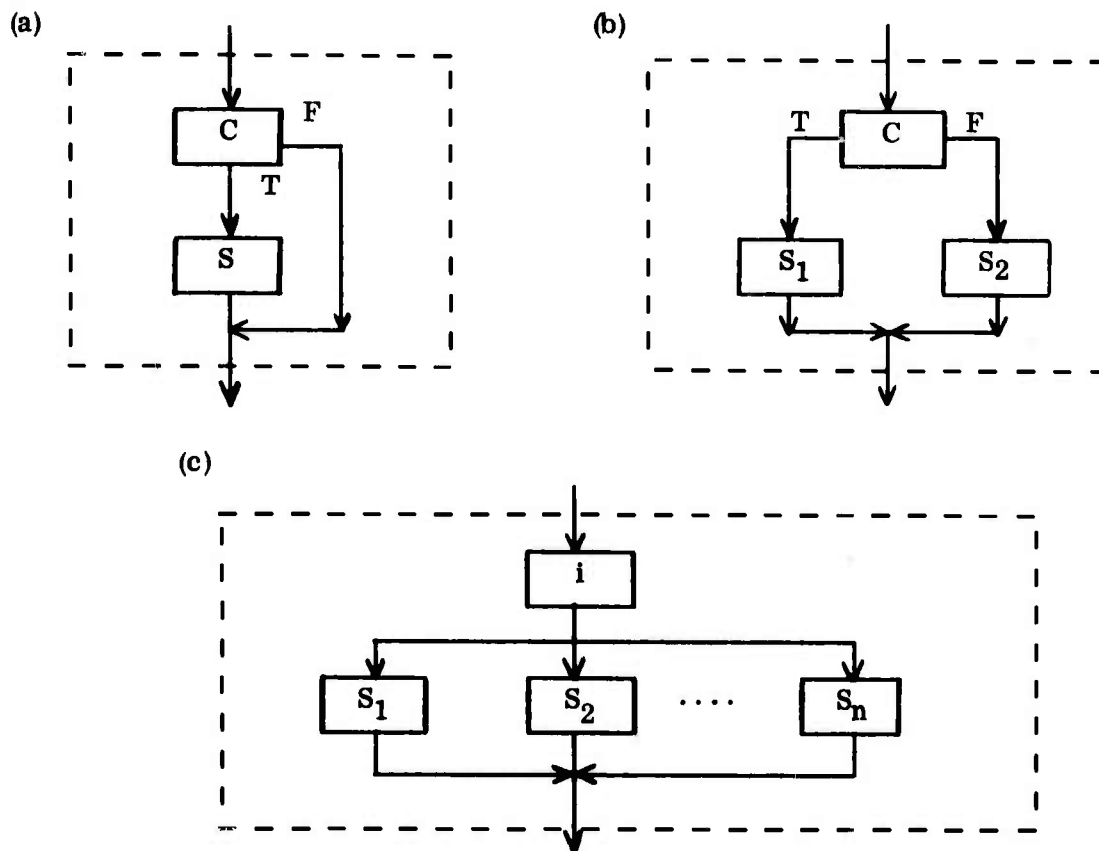
Flow charts:



3) Selection

Program forms: (a) if C then S;
 (b) if C then S_1 else S_2 ;
 (c) case i of $\{S_1, S_2, \dots, S_n\}$;

Flow charts:



The dashed boxes around the statement flow charts are meant to emphasize the fact that the sequencing construct is treated as a unit.

Two programs developed as part of the PDP-11 simulation effort illustrate this principle. Figure 2.13 shows a flow diagram for the program PDP11SIM (Figure 2.10). The program can be considered to consist of a single statement which repeats a compound statement. When the program PDP11SIM is activated, the instruction counter for this outer level program is set to point to this single statement; when it steps past it, the program is over. Progressing inward, the compound statement,

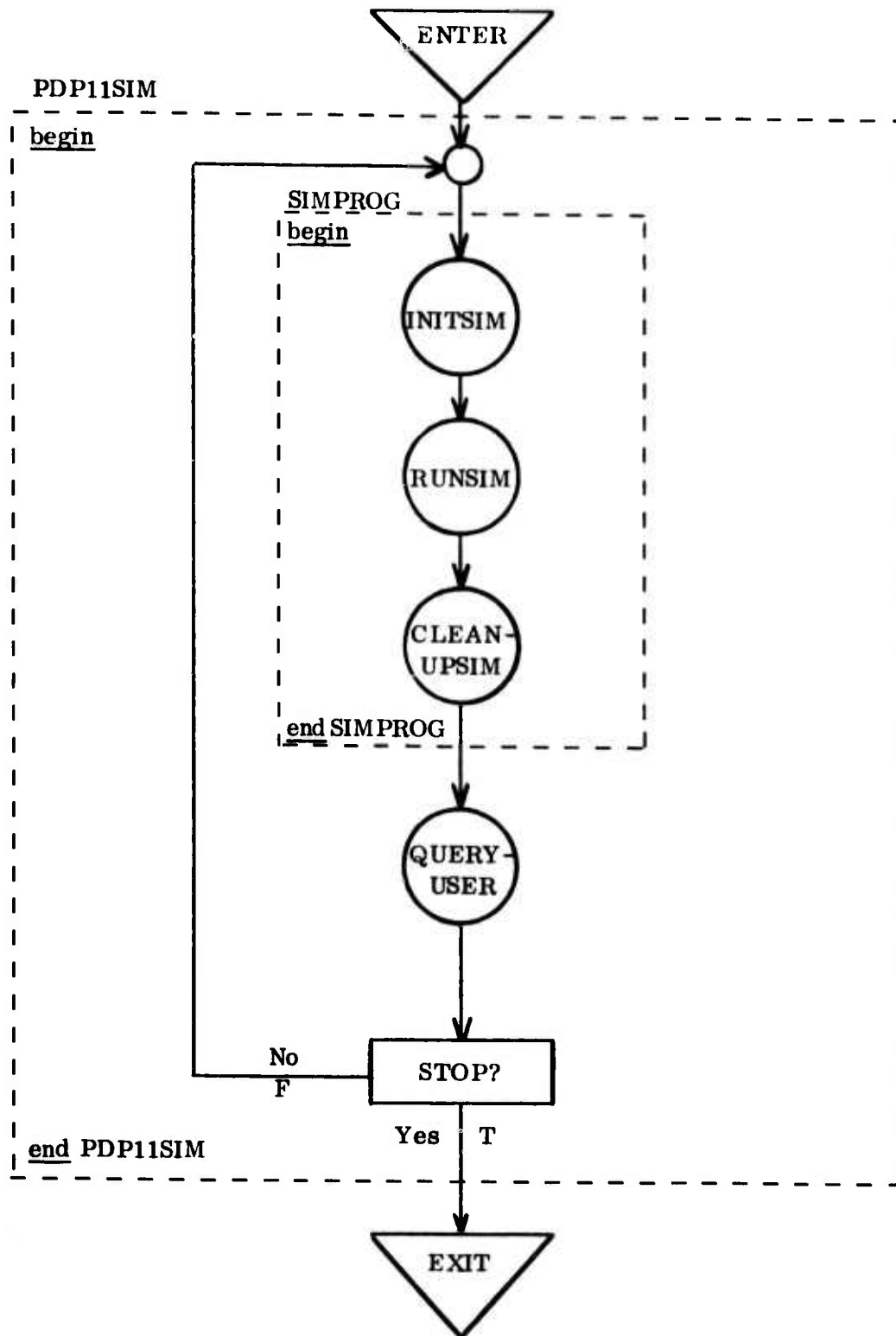


Figure 2.13. Flow Diagram for the Program PDP11SIM

named SIMPROG, consists of three program "calls" enclosed in the brackets begin and end. This compound statement illustrates a simple ordering of statements. It also illustrates again the use of a simple sequence as a single entity, for SIMPROG is itself made up of a sequence of three program calls or stubs.

Figure 2.14 shows a more complex combination of the control structures which still preserves the sequential ordering of a program, RUNSIM. The text of the program, Figure 2.15, shows the use of the control structures.

- d. Principle #4. Develop the control structures before the development of other program details.

An examination of the text of the two programs, PDP11SIM and RUNSIM (Figures 2.11 and 2.15) used to illustrate the earlier principles shows the general lack of programming detail in the upper levels of the program. Using only the stubs of programs and the controls which govern their activation, the design has progressed from control of the simulator through the control of the cycling of the simulated machine.

Before any of the details of the PDP-11 actually enter into the program, the sequencing controls for the first four levels of the program are fully developed. Until the program FINDFORMAT is composed at LEVEL5 of the program, the simulation could be of an IBM 360 or any other computer.

The actual details of how the programs which give meaning to the stubs operate are deferred until the control structure is well developed.

- e. Principle #5. Defer decisions about data representation and other details as long as possible.

This principle is also difficult to illustrate, since it too consists of not doing something as long as possible. In the program outline developed for the PDP-11 simulation program in the next section, the program is organized into major parts. The sequencing controls that satisfy the major design constraints are provided without a need to make any decisions regarding the representation of PDP-11 instructions, memory, or any other aspect of the target machine until the fifth level of the program.

In deciding about data for the PDP-11 simulation, it is clear that the instructions for the simulated machine will have to be available for the part of the program

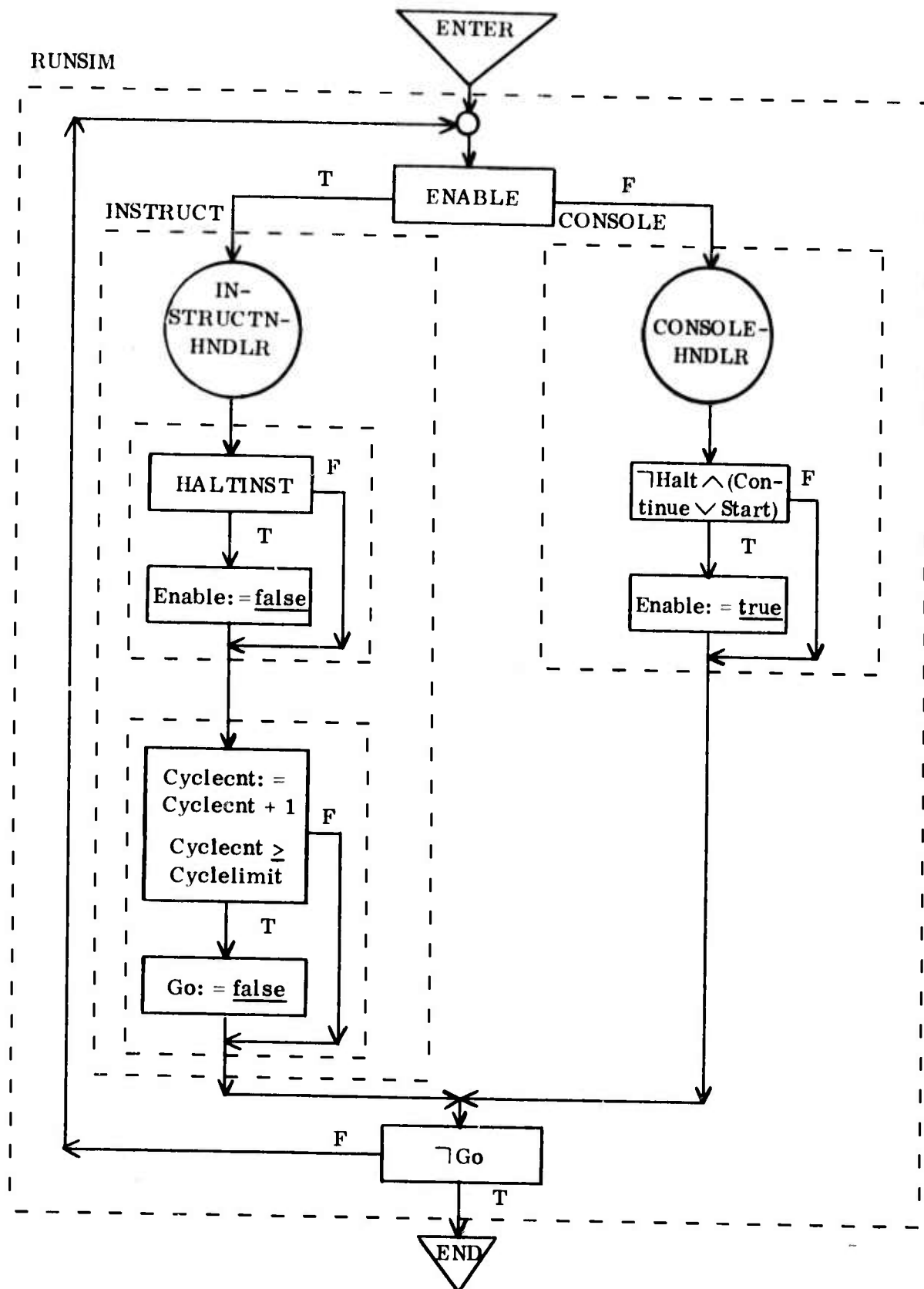


Figure 2.14. Flow Diagram for the Program RUNSIM

```

RUNSIM (Cyclelimit): begin
comment    control a simulation run. A run is repeated execution of console or
              instruction cycles. A run is terminated by exceeding the Cyclelimit
              or by direction of the user;

boolean    Enable: = false; Halt: = true; Go: = true;
integer    Cyclecnt: = 0;

repeat
  begin
    comment    provide connecting link between the two kinds of cycles,
                  as long as Go is true;

    if Enable then
      INSTRUCT: begin
        boolean    Haltinst: = false;
        INSTRUCTNHNDLR (Haltinst);
        if Haltinst then Enable: = false
        Cyclecnt: = Cyclecnt + 1;
        if (Cyclecnt ≥ Cyclelimit) then Go: = false;
        end INSTRUCT;
      else
        CONSOLE: begin
          boolean    Continue: = false; Start: = false;
          CONSOLEHNDLR (Halt, Continue, Start, Go);
          if ( $\neg$ Halt  $\wedge$  (Continue  $\vee$  Start)) then Enable: = true;
          end CONSOLE;
        end;
      end;
    until  $\neg$ Go;
  end RUNSIM;

```

Figure 2, 15. Text of RUNSIM

that executes the instruction cycle. It is equally clear, however, that the instructions being simulated must be entered or read as data outside of this function.

Since the data for the simulation is entered as a separate step, and must also be accessible to the step that simulates the machine, storage for that data must be reserved outside of the scope of either step. In keeping with the concept of abstract machines, a program at a very low level, used by both branches of the program organization, handles the data representation and the space. In this study, the Algol own array construct is used to control the scope of access to the variable. Thus the need for data can be recognized at the point where it is required without necessarily defining in detail how it is to be represented. In the specific example of PDP-11 instructions, it is necessary to recognize their existence at the point where they would be entered as data. The decision on how to represent PDP-11 memory words in the simulator can be put off to the point where this information is needed to continue program development (e.g., in the simulated instruction processing). At this point one can determine the kind of machine the simulator is to run on and determine whether it is possible to represent the PDP-11 memory as one word per word of the host machine, or whether a packed form of two words per word of host machine memory is needed.

SECTION III

APPLICATION OF THE PRINCIPLES OF STRUCTURED PROGRAMMING

1. INTRODUCTION

A program was written to demonstrate and evaluate the use of Structured Programming. The problem used for the demonstration was selected on the basis of three criteria: the problem had to be large enough to provide an adequate test of the use of Structured Programming; it had to be small enough to be realizable within the constraints of the study effort; the essence of the problem had to be familiar to the investigator to avoid distractions from the approach taken to the program.

2. TEST PROBLEM

The problem selected for use in the study of Structured Programming was: "Build an interpreter for the DEC PDP-11."

The control structure for an interpreter, with its continuous operation, is deemed to be more appropriate to the avionics environment than the sequential problems that have appeared in the literature to date. The problem models, in a simple way, a control environment involving continuous operation, data generated internally or acquired by "sensing" its environment, and the selection of certain actions based on an analysis of the data. In addition, it meets the three criteria enumerated above.

A further interest in the problem is based on the potential use of simulators in microprogrammed processors for avionics systems. This problem involved interpretative execution of programs similar to that done for emulation, and could serve as a model for emulator design.

The PDP-11 is a 16 bit binary computer. A variable length instruction format is used to provide optional zero, one and two operand addressing. Multiple modes of addressing are provided with optional direct and indirect addressing. Indexing of operand addresses is provided together with an option for automatic incrementing and decrementing of the index. Seven general registers are provided for use by the programmer. In addition to the variable length of the instructions, there is also a variable format for the instructions which provides for an expanded set of instructions. Input-Output operations are controlled through registers that are within the normal address space of the machine. The control of peripherals

involves two registers, one for data and one for the status of the device. The status words are used to direct the input-output operation. These registers may be used as operands with any of the instructions. Specific details of the machine operation are available from Digital Equipment Company. ²⁰

3. THE PROGRAM

The program is designed for execution from a remote terminal of a time-sharing system. Facilities of the time-sharing terminal are used to simulate the console switches of the PDP-11. Detailed internal timing aspects of the bus actions and I/O interfaces are not maintained; however, simulated I/O operations are activated and controlled through program accesses to related addresses in the simulated memory. The operations are carried out as movements between the simulated memory and file space in the host machine. The files are filled from the terminal if the file is empty and input is required by the program.

Program loading into the simulated memory is carried out either through the simulated console actions or through the execution of the PDP-11 bootstrap loader on the simulated machine. Other software aids must be provided in a similar manner.

The simulator maintains a cycle count and limit as a protection against the occurrence of an infinite loop in a program being interpreted. The user is given the capability to specify both the memory size and the I/O device compliment.

a. Program Text and Flow Diagrams

The programs presented here are a subset of the programs developed in composing a program to illustrate the principles of Structured Programming. These principles are followed in the program construction. The text for the program begins at the level which contains a single program, PDP11SIM. This program is the entry point into the simulation and is a functional specification for the operation of the system. The program is constructed of a set of controls used to organize the activation of other programs, represented by stubs, or program "calls." These programs correspond to the subtasks identified during analysis by problem decomposition.

The program PDP11SIM is given in Figure 3.1 (flow diagram in Figure 3.2). There are three layers of decomposition of the problem represented in this one program. The outermost decomposition is represented programmatically by the iteration statement

repeat begin ... end until Stop

```

PDP11SIM: begin
comment    establish iterative control to allow repetition of simulator for
multiple runs;

boolean    Stop;

repeat
    begin
        SIMPROG: begin
        comment    perform a single simulation run;
        integer    Cyclelimit;
        comment    supplied by user to limit the number of
machine cycles executed;

        INITSIM (Cyclelimit);
        comment    initialize simulator to appropriate I/O and memory
configuration. Get value for Cyclelimit, let user
load peripheral files;

        RUNSIM (Cyclelimit);
        comment    perform a run on simulated machine. Use Cyclelimit
to control against infinite loop in user program;

        CLEANUPSIM;
        comment    cleanup any residual I/O left when RUNSIM quit;

        end SIMPROG;

        QUERYUSER (Stop);
        comment    talk to user and find out if should go again;

        end;
    until Stop;

comment    end of controlled loop for repeated simulations;

end PDP11SIM;

```

Figure 3.1. Text of PDP11SIM

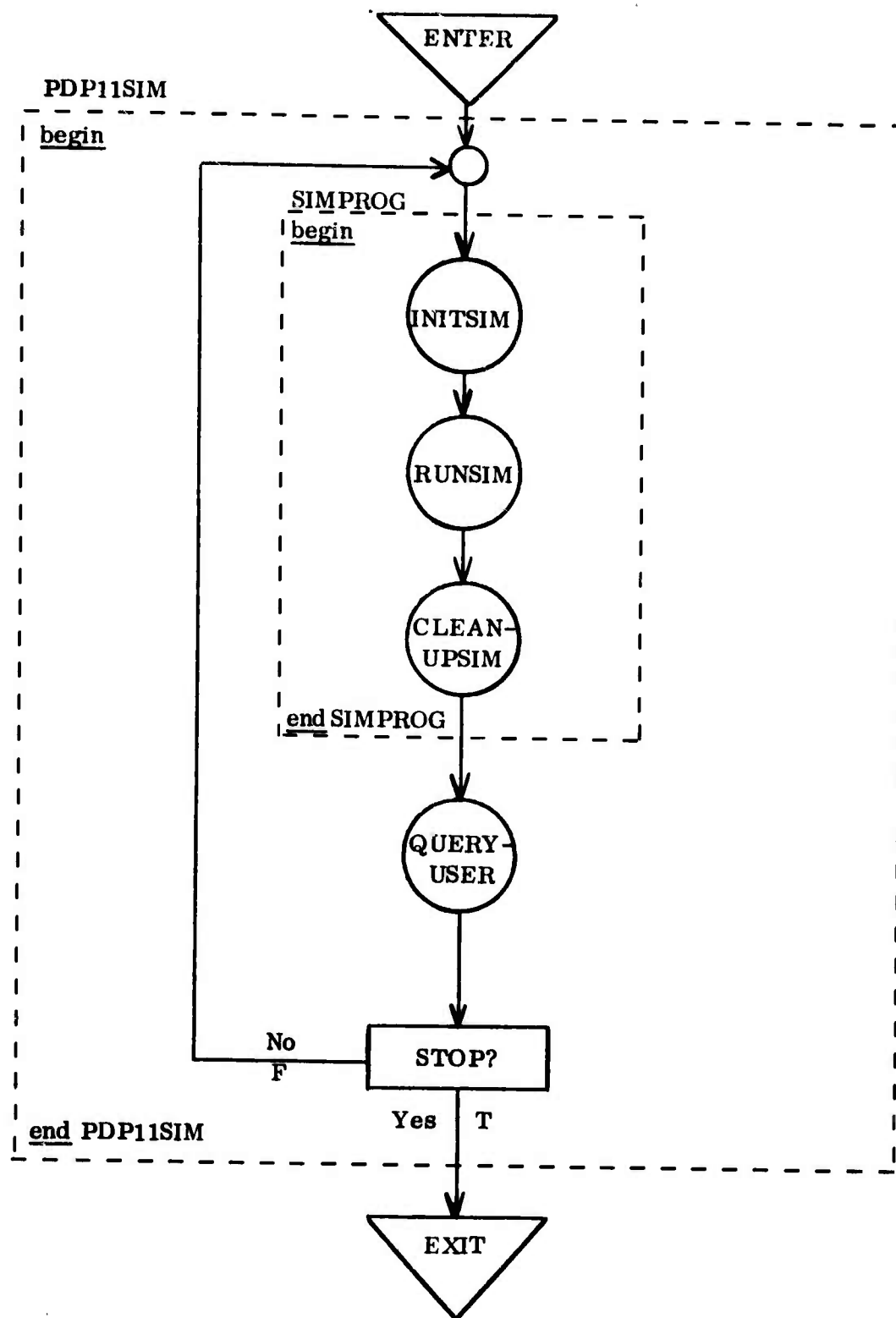


Figure 3.2. Flow Diagram for the Program PDP11SIM

The use of the simulator can involve multiple operations of the simulation program, possibly with different configurations of memory or I/O. Since the number of simulations desired at any one time may vary, it is not practical to represent this decomposition in any way except through a repetition statement. A control variable, Stop, used to terminate the repetition is defined in the program.

The subtask controlled by the repetition is itself partitioned into two subtasks, identified by the programs SIMPROG and QUERYUSER. The purpose of SIMPROG is to perform a simulation, and the purpose of QUERYUSER is to obtain a value for Stop from the user of the program. A third layer of decomposition is shown in the subdivision of SIMPROG into the sequential steps necessary to perform initiation of a simulation, a simulation, and a clean-up of a simulation, respectively.

b. Second Level of Program Organization

The "top" level program, PDP11SIM, defers the details of the program actions for specification in lower level programs. PDP11SIM specifies the exact conditions under which each of the four (4) programs (INITSIM, RUNSIM, CLEAN-UPSIM, and QUERYUSER) will be activated for execution. All of these programs are candidates for inclusion in the next level of the program organization. The definition of the program QUERYUSER is deferred to a lower level because it is very much dependent upon a particular installation's run time environment and because it is very close to being "completely" defined in some languages.

The three programs placed at the second level are all derived from the program SIMPROG, contained within PDP11SIM. These programs perform the three subtasks required to carry out a single execution of the simulation. Of the three programs at this level, the program RUNSIM, which actually performs the simulation, is presented in this section.

RUNSIM, as can be seen from its text (Figure 3.3), is, like PDP11SIM, a very simple program. It also consists of the repeat of a compound statement until a control variable, Go, has a value of false. This loop simulates the continuous cycling of a computer. Each time the simulated computer cycles it selects the proper task to be performed, as indicated by the state of a control variable, Enable. Whenever Enable is set to true, an instruction is processed. If Enable is false, a console action is initiated. The nesting of these structures in the program is clearly shown in the flow diagram (Figure 3.4). The details of the program were developed until the facilities for assigning values to Go and Enable were specified.

```

RUNSIM (Cyclelimit): begin
comment    control a simulation run. A run is repeated execution of console or
              instruction cycles. A run is terminated by exceeding the Cyclelimit
              or by direction of the user;

boolean    Enable: = false; Halt: = true; Go: = true;
integer    Cyclecnt: = 0;

repeat
  begin
    comment    provide connecting link between the two kinds of cycles,
                as long as Go is true;

    if Enable then
      INSTRUCT: begin
        boolean    Haltinst: = false;
        INSTRUCTNHNDLR (Haltinst);
        if Haltinst then Enable: = false
        Cyclecnt: = Cyclecnt + 1;
        if (Cyclecnt ≥ Cyclelimit) then Go: = false;
        end INSTRUCT;
      else
        CONSOLE: begin
          boolean    Continue: = false; Start: = false;
          CONSOLEHNDLR (Halt, Continue, Start, Go);
          if ( $\neg$ Halt  $\wedge$  (Continue  $\vee$  Start)) then Enable: = true;
          end CONSOLE;
        end;
      until  $\neg$ Go;
    end RUNSIM;
  
```

Figure 3.3. Text of RUNSIM

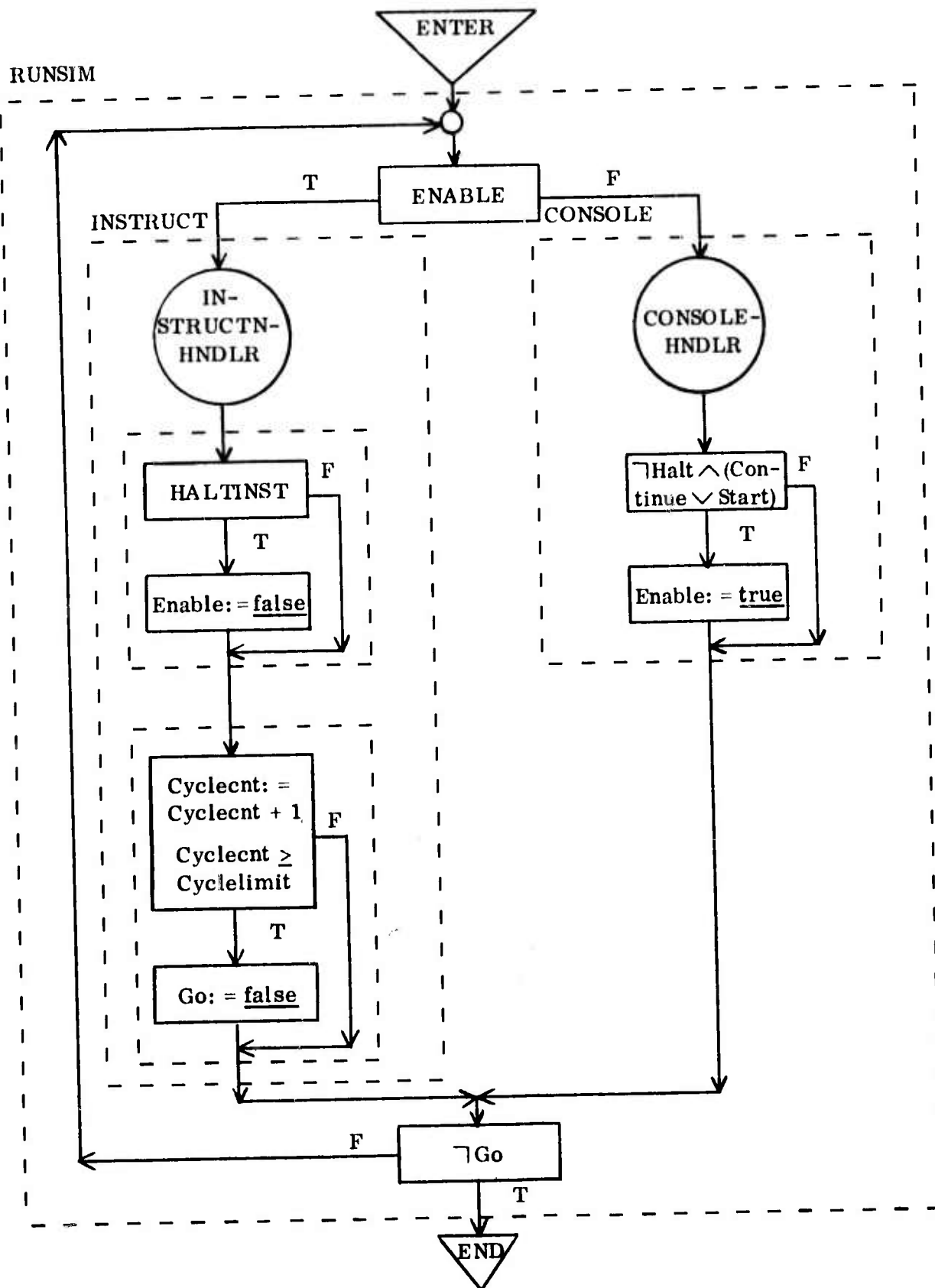


Figure 3.4. Flow Diagram for the Program RUNSIM

c. INSTRUCTNHNDLR, a Third Level Program

The program INSTRUCTNHNDLR is a subtask of the program RUNSIM. The task addressed by this program is the organization of what is often called an instruction cycle. A set of subtasks are designated which, when carried out in order, will process an instruction. The sequential organization of these tasks is seen in the program text (Figure 3.5) and the flow diagram (Figure 3.6).

This program reflects the transition from the control of the simulation to a closer involvement with the PDP-11 machine organization. The impact of the hardware organization is reflected in the functions of three of the five subtasks formulated. Often, the first phase of instruction processing is interrupt processing. In the PDP-11 this function is performed in a slightly more general way, although the result is the same net effect. The name ISELECT indicates the change of emphasis from the more familiar interrupt toward the more general selection between alternative sources for the next instruction.

Normally, an instruction is decoded after being brought to the processor. Because of the variable instruction format and the variable number of operands used for instructions on the PDP-11, the emphasis shifts from a decoding operation to one of format identification. The procedure FINDFORMAT reflects this shift. An operand fetch requirement is automatic in many computers. The conditional execution of the subtask, OPFETCH, is a reflection of the large number of instructions which do not require normal operand processing. After the operand processing is accomplished, the instruction can be executed by the program IEXECUTE.

d. FINDFORMAT and IEXECUTE - Two Fourth Level Programs

The programs used to develop the program INSTRUCTNHNDLR are all collected into the fourth level of the program organization. Two of these programs are closely related. They are both presented here. The text of the first of these programs, FINDFORMAT, is presented in Figure 3.7; the flow diagram in Figure 3.8. It is also a complex structure, set up as a sequence of three simple operations (REMOVEBYTE, GETOPFIELD, and OPFIELD). The first two of these operations use stubs to defer any detailing of the functions required. The third divides the task of identifying the opcode into two parts. For those instructions that use an extended opcode field ($OP = 0$), a stub (FINDONEOP), is used to defer the specification of the processing details. Those instructions which use only the normal opcode field for identification are decoded for illegal instructions and to separate addition and subtraction through the use of a local program (TRY2). The name TRY2 indicates that if the instruction is valid, it will involve two operands. Within the program TRY2 the two machine instructions, add and subtract, are separated. These two instructions are distinguished by the state of the byte indicator.

```

INSTRUCTNHNDLR (Haltinst): begin
comment      process a single machine instruction from the program being
                pseudo executed;

boolean      Byte, Haltinst, Sub;

integer      Source, Instruction, Destination, Adsource, Ades, J: = 0;
comment      Instruction, Source and Destination are 6 digit octal numbers
                where value is less than  $2^{16}$ . J is an integer between 0 and 2.;

ISELECT:
comment      check for interrupts, if priority of interrupt exceeds that of
                processor then save current instruction address and contents
                of status register in stack. Set up new instruction pointer and
                status from interrupt vector;

IFETCH (Instruction);
comment      get instruction from memory and return it as a result;

FINDFORMAT (Instruction, Byte, J, Sub);
comment      with instruction as an input, do a partial decode to determine the
                number of operands required; isolate the Byte indicator in the
                boolean Byte. If instruction is a subtraction set Sub to true, set
                J equal to number of operands required;

if (J  $\neq$  0) then OPFETCH (Instruction, Byte, J, Source, Destination, Adsource, Ades);
comment      if any operands are required, then place them in Source and/or
                Destination and some effective address;

IEXECUTE (Instruction, Byte, J, Sub, Source, Destination, Adsource, Ades, Haltinst);
comment      perform the proper instruction and return result to storage if appro-
                priate. If instruction is a Halt, then set Haltinst to true, else set
                false.

end INSTRUCTNHNDLR;

```

Figure 3.5. Text of INSTRUCTNHNDLR

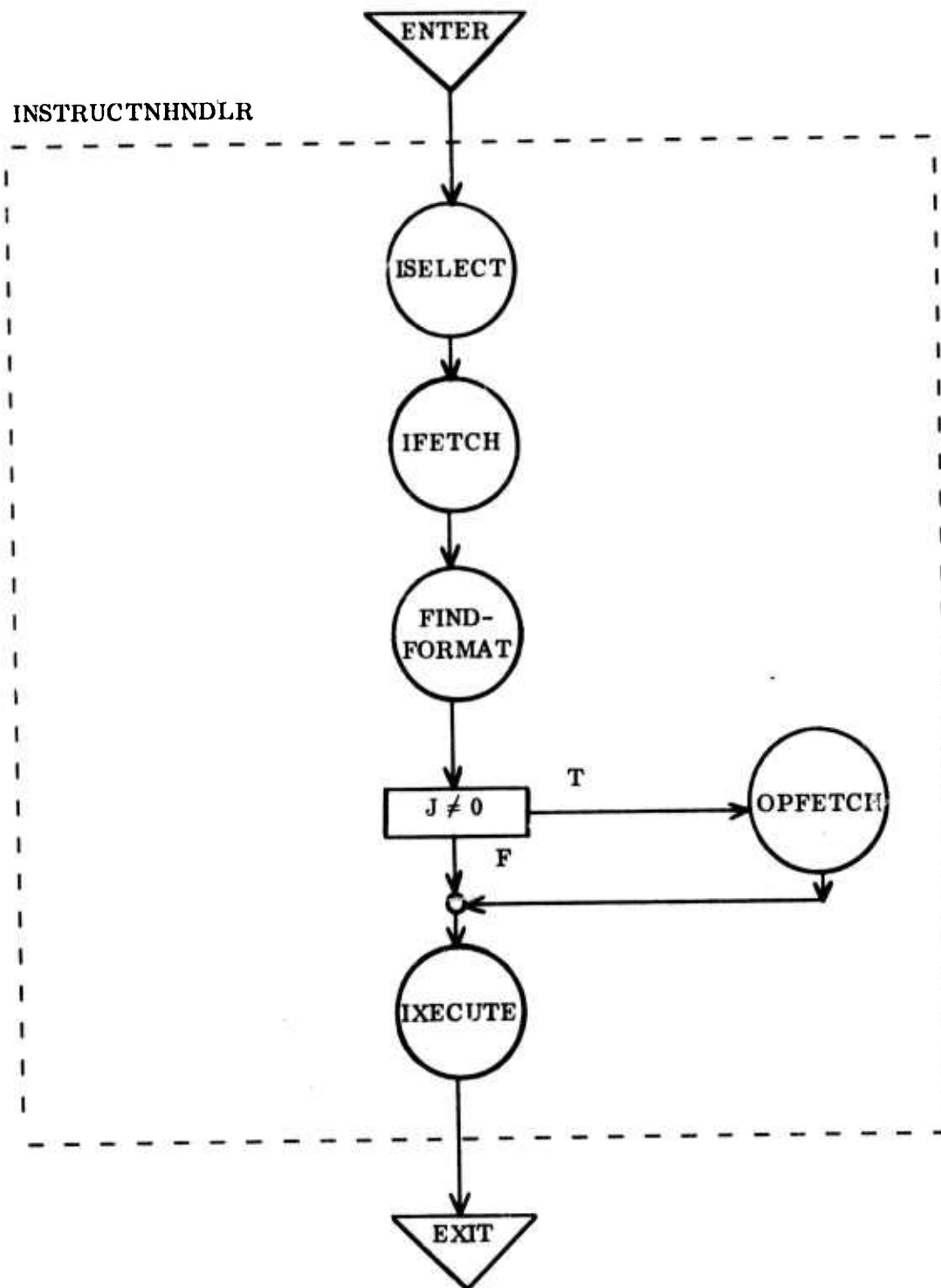


Figure 3.6. Flow Diagram for Program INSTRUCTNHNDLR

```

FINDFORMAT (Instruction, Byte, J, Sub): begin
comment      given the instruction as input, determine the number of
                operands required for the instruction, J; isolate the Byte
                indicator and set the boolean Sub to indicate subtraction;
                bits in the input word are numbered from high to low (15 ... 0);

integer      Op;

REMOVEBYTE (Instruction, Byte);
comment      bit 15 indicates a byte operation; move bit 15 to Byte. Remove
                bit 15 from Instruction;

GETOPFIELD (Instruction, Op);
comment      move bits 14, 13, 12 into Op;

OPFIELD: begin
comment      determine number of operands required;

                if (Op = 0), then FINDONEOP (Instruction, Byte, J);
                else
                        TRY2: begin
                                comment either double operand or illegal;
                                if (Op ≠ 7), then
                                        LEGAL: begin
                                                if (Op = 6), then
                                                        ADDOP: begin
                                                                comment (Op = 6) ∧ ¬Byte is add.
                                                                (Op = 6) ∧ Byte is subtract.
                                                                if Byte, then
                                                                        SUBTRACT: begin
                                                                                Byte: = false
                                                                                Sub: = true
                                                                        end SUBTRACT;
                                                                end ADDOP;
                                                        J: = 2;
                                                end LEGAL;
                                        end TRY2;
                                end TRY2;
                        end OPFIELD;
end FINDFORMAT;

```

Figure 3.7. Text of FINDFORMAT

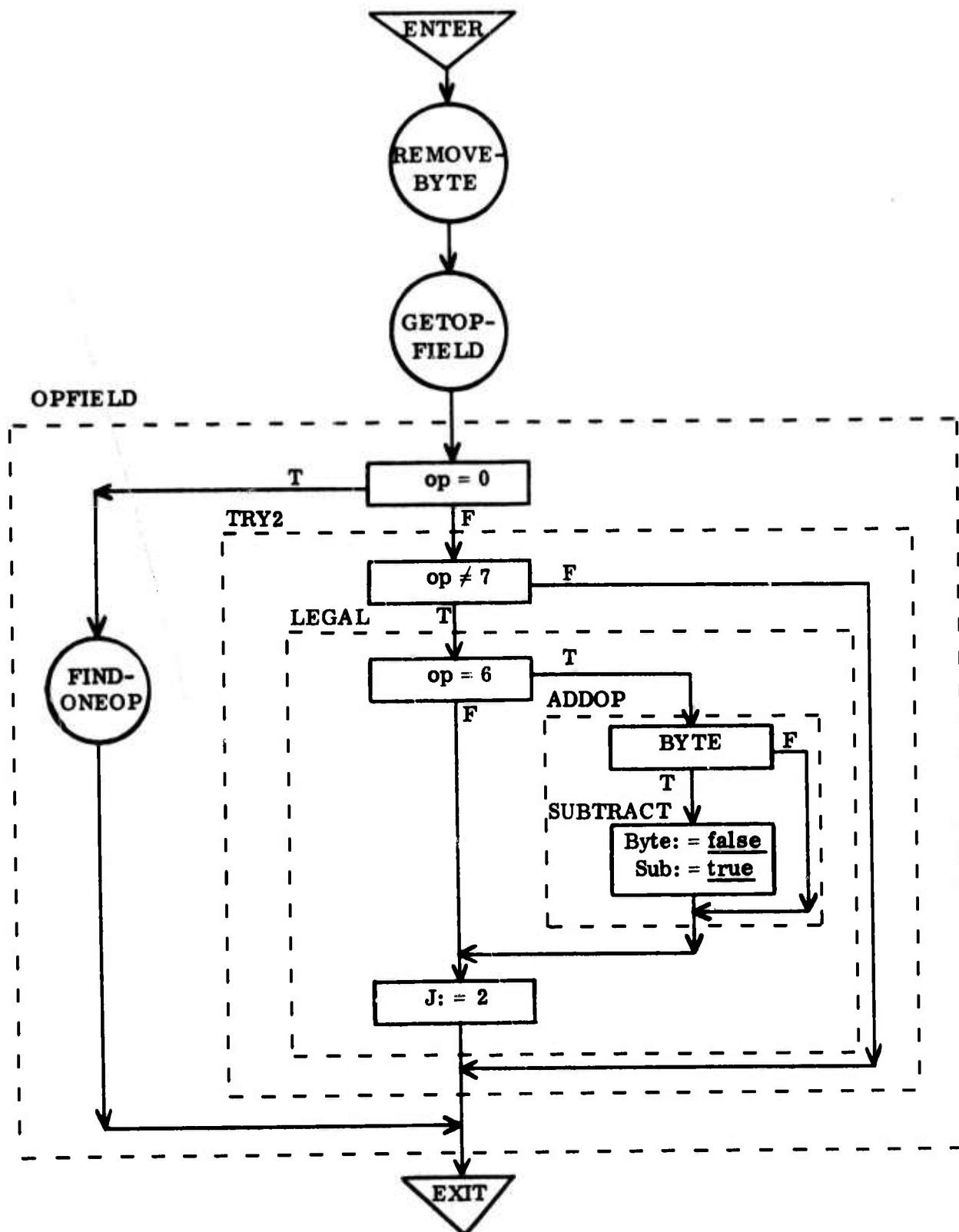


Figure 3.8. Flow Diagram for the Program FINDFORMAT

IXECUTE is the second program of the fourth level programs presented. This is the program which actually carries out the PDP-11 instruction. A case statement is used to separate the instructions into one of three classes of instruction: no operands, one operand and two operands.

After the instruction is performed, the result is stored into the destination address for those instructions that require the result. The text of the program is given in Figure 3.9 and the flow diagram in Figure 3.10.

```

IXECUTE (Instruction, Byte, Sub, Source, Destination, Adsource, Ades,
        Haltinst): begin
comment    use J values to separate the instructions according to a number
              of operands. Source and Destination contain values at this point;
              the only output of this program is Haltinst;

case       J of      { OPNONE (Instruction, Byte, Haltinst),
                        OPONE (Instruction, Byte, Destination),
                        OPTWO (Instruction, Byte, Sub, Source, Destination)};

if J  $\neq$  0, then STORERESULT (Destination, Ades);
end IEXECUTE;

```

Figure 3.9. Text of IEXECUTE

e. LEVEL5 Programs

There are outstanding stubs (i.e., have not been specified with a program) from levels 1, 2, 3 and 4. By choice, LEVEL5 programs are limited to those stubs from LEVEL4. The program stubs for REMOVEBYTE and GETOPFIELD are not specified at LEVEL5 because of their use in other programs at lower levels. There are two sample programs from LEVEL5 presented here, FINDONE-OP and OPNONE.

The program FINDONEOP, Figures 3.11 and 3.12, is stubbed in FIND-FORMAT. It is the most complex program presented. The Algol procedure declaration is used to organize the parts of the program to improve the readability.

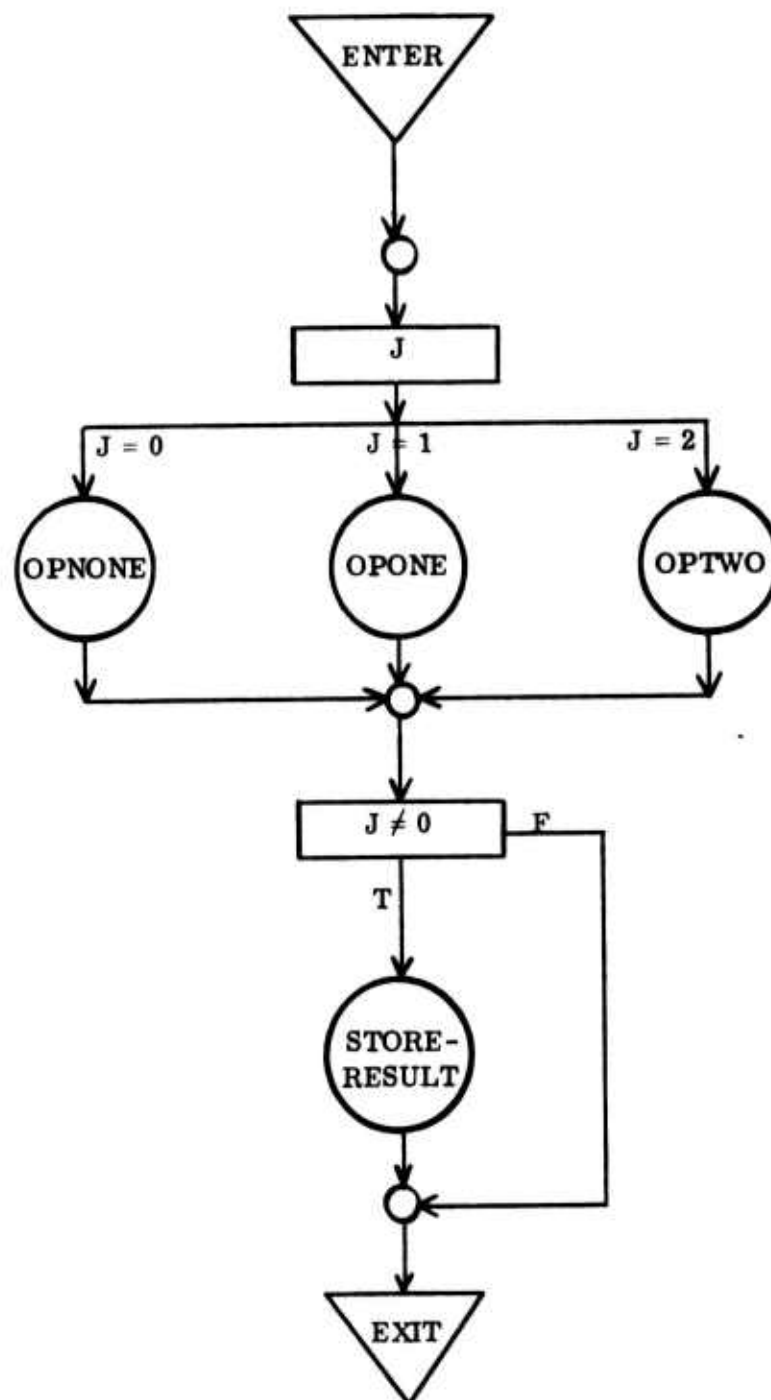


Figure 3.10. Flow Diagram for the Program IEXECUTE

```

FINDONEOP (Instruction, Byte, J): begin
comment    if the instruction involves a single operand, then set J: = 1
            else set J: = 0;

integer    Hisource, Losource, Hides;

procedure  MORETEST: begin
comment    Losource < 3 then shift else not one operand;
GETLOSOURCE (Instruction, Losource);
comment    get bits 8, 7, 6;
if (Losource < 3) then J: = 1;
end MORETEST;

procedure  TESTBYTE: begin
comment    if not byte then JSR else traps and not one operand;
if non Byte then J: = 1;
end TESTBYTE;

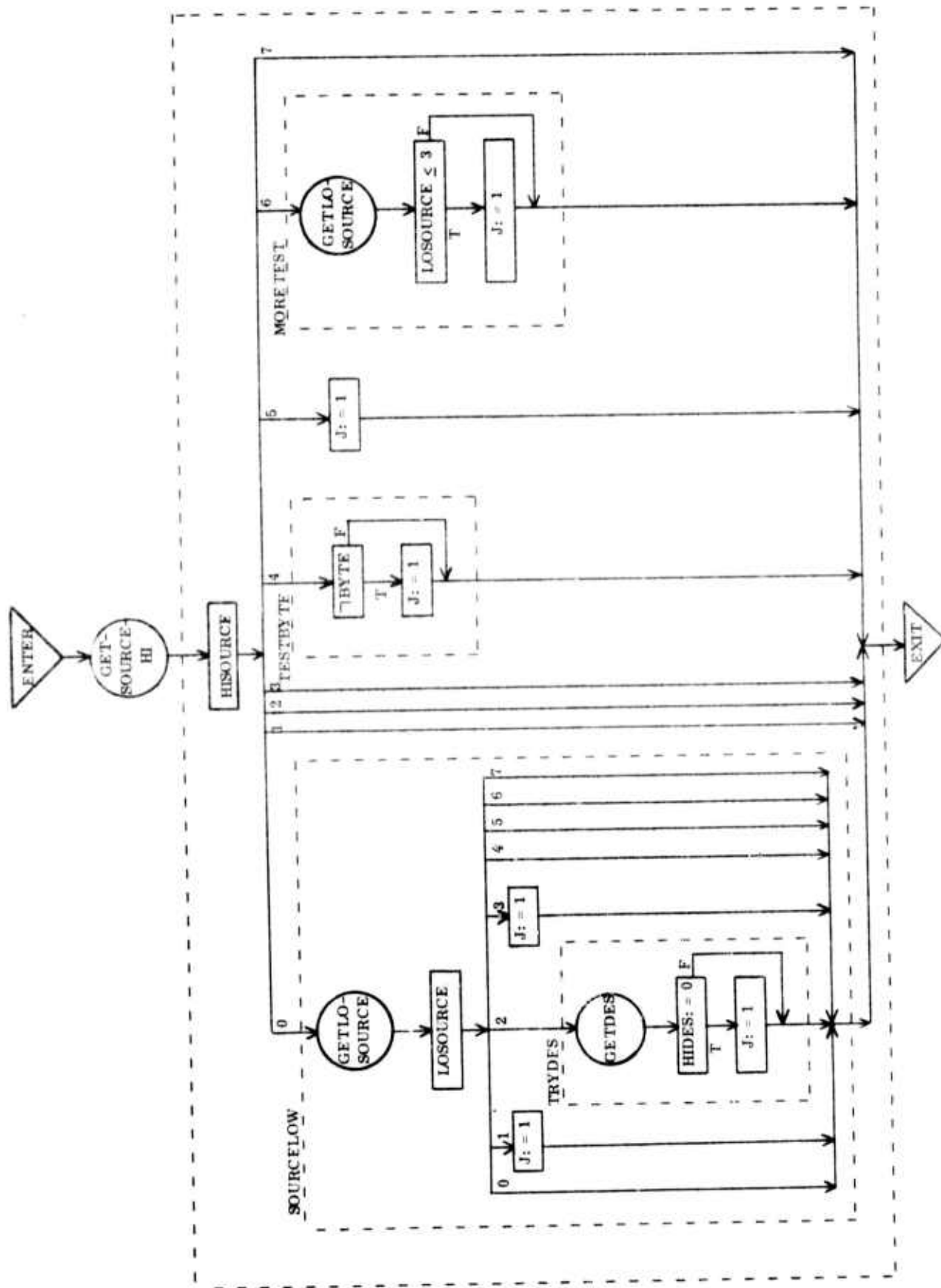
procedure  TRYDES: begin
comment    if Hides (bits 5, 4, 3) = 0, then in RTS and needs operand;
GETDESHI (Instruction, Hides);
if (Hides = 0) then J: = 1;
end TRYDES;

procedure  SOURCELOW: begin
comment    if Losource = 1/2/3 a possible single operand instruction;
GETLOSOURCE (Instruction, Losource);
case        Losource of { , J: = 1, TRYDES, J: = 1, .... } ;
end SOURCELOW;

PROGRAM: begin
GETSOURCEHI (Instruction, Hisource);
case        Hisource of { SOURCELOW, ..., TESTBYTE, J: = 1, MORETEST, . };
end PROGRAM;
end FINDONEOP;

```

Figure 3.11. Text of FINDONEOP



Despite the complexity of the nested statement, it is constructed first as a simple ordering of two tasks: 1) isolate the high byte of the source field of the instruction; and 2) assign a value to J based upon this high byte. Whenever the high byte does not completely determine the required number of operands, other fields are examined. High source bytes having the values of 1, 2, 3 or 7 set J to 0. A high source of 5 results in J set to 1. Values of 0, 4 and 6 use a further decoding of the instructions to determine the proper settings for J.

The program OPNONE (Figures 3.13 and 3.14) offers another contrast in the possible complexity of a program, constructed of an ordered sequence of computational events. The purpose of the program is to direct each of the instructions which require no operands to the program which will simulate its effect. Thus the program completes the decoding of the instructions that do not require an operand for their execution.

The decoding process is carried out by a set of six nested selection statements. Within the second alternative of the sixth case a case statement is used to select among six alternatives based on the value of the instruction.

4. SAMPLE PROGRAMS AS AN ILLUSTRATION OF THE PRINCIPLES

The text of the program to simulate the PDP-11 begins with a program, PDP11SIM (Figure 3.1) which specifies a solution to the problem. This is in keeping with the first principle of Structured Programming. The principle calls for the development of programs from the "top-down" in a way which reflects a "top-down" decomposition of the problem.

Those texts presented in the preceding part of this section are representative of this form of program organization. The first program, PDP11SIM, presents enough detail to completely specify the controls required to allow multiple runs of a simulator. Each of the simulations are performed by the program SIMPROG, specified within the text of PDP11SIM. A second program, QUERYUSER, is identified as the source of the value of the control variable Stop, used to terminate the program.

The program SIMPROG is specified to be composed of three parts, and the labels used for the parts indicate the function to be performed by them (INITSIM, RUNSIM and CLEANUPSIM). The order in which they are to be carried out is also defined in the program.

```

OPNONE (Instruction, Byte, Haltinst): begin
comment    form subsets of the no operand instruction for further decoding and
              execution. Byte and Instruction are the basis for set identification;

if (Instruction  $\geq$  6400) then ILLEGAL;
  else begin ONE: if (Instruction  $\geq$  4400) then TRAP;
    else begin TWO: if (Instruction  $\geq$  4000) then EMT;
      else begin THREE: if (Instruction  $\geq$  400) then CONDBRNCH;
        else begin FOUR: if (Byte) then CONDBRNCH;
          else begin FIVE: if (Instruction  $\geq$  240) then CONDITION;
            else begin SIX: if (Instruction  $\geq$  6) then ILLEGAL;
              else case: Instruction of { HALT, WAIT, RTI, BPT, IOT, RESET };
                end SIX;
              end FIVE;
            end FOUR;
          end THREE;
        end TWO;
      end ONE;
    end OPNONE;

```

Figure 3.13. Text of OPNONE

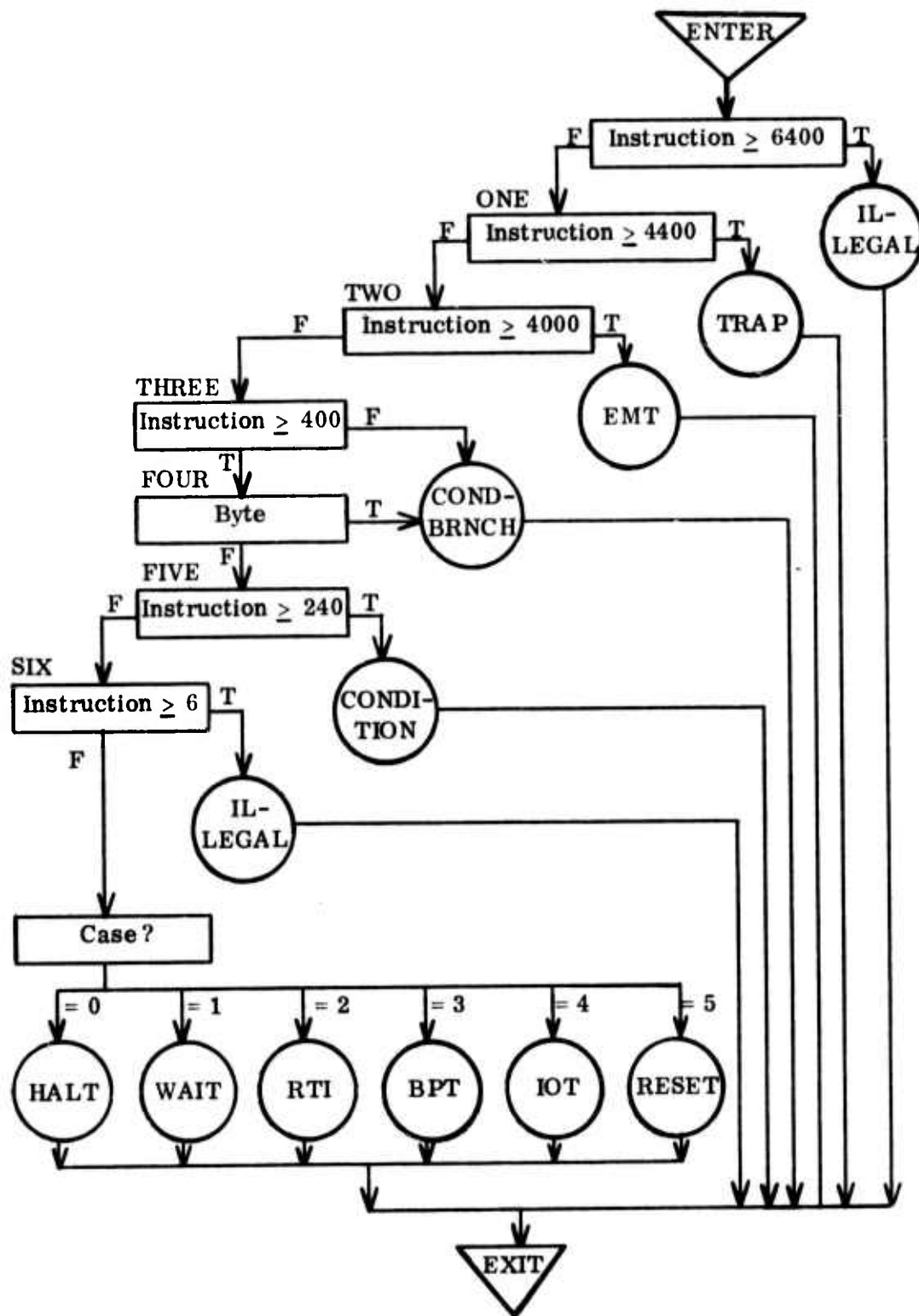


Figure 3.14. Flow Diagram for the Program OPNONE

This one program then presents in cameo the organization of the problem into the top-down program. An examination of the programs found in the succeeding levels of the program organization shows further formulation of the details involved in the solution of the problem.

The program RUNSIM (Figure 3.3), for example, illustrates the control of the actual simulation process. RUNSIM is activated after the initialization of the simulation is carried out. It is terminated by the state of a control variable, Go. The simulation process is performed by the repetition of an unnamed process. Each time this process is repeated, an action is selected based on the state of the variable Enable. The program RUNSIM is developed in sufficient detail to specify how the values are assigned to both of the control variables (Go and Enable). Thus a net effect which simulates the cycling of a computer is achieved.

The "top-down" development is continued at the lower levels through the specification of the actions represented by INSTRUCTNHNDLR and CONSOLEHNDLR.

Each level of the program organization specifies in greater detail action identified in a predecessor program. Programs are presented for each subtask identified in the analytical process of problem decomposition.

Principle Two specifies that the program should be organized into distinct levels forming an ordered series of intermediate systems of programs. The programs used in the example have been identified as being assigned to various levels. Beyond observing the required ordering between a stub and its definition, the programs have been grouped into levels in a way which isolates specific aspects of the program.

The motivation for some of the grouping has already been discussed. The first level of the program organization contains the single program, PDP11SIM, which specifies the problem solution. At the second level, a choice is made not to include QUERYUSER because it does not deal with the problem of simulation. The stub is left undefined until LEVEL8 of the program organization. LEVEL2 consists of the programs INITSIM, RUNSIM and CLEANUPSIM, all of which are stubs in SIMPROG.

At LEVEL3, only two programs are included, INSTRUCTNHNDLR and CONSOLEHNDLR. This decision reflects the intention to isolate in the levels as much of the interpretation process as possible. At LEVEL4 the parts of INSTRUCTNHNDLR are collected and the parts of CONSOLEHNDLR are excluded. This separation was based upon the differences between the console simulation in this interactive program and any microprogrammed implementation of a console. There is an intuitive feeling that an emulation of a PDP-11 could use the instruction processing program as its basis, but would not be likely to use the console processor.

Both the layers in the tree of decomposition and the levels of the program text form a hierarchy; they are related to each other. A subtask appears in layer "n" of the tree of decomposition because it is a part of a task in the layer "n - 1". A program is assigned to a level based on its predecessor and also on its having a common level of abstraction with other programs on that level. A sub-system of programs is assembled in a level to facilitate testing of the programs and to isolate the impact of adaptations and changes to the program.

The third principle of Structured Programming deals with the allowable set of control structures. In particular it emphasizes the simple ordering of a sequence of program steps. The purpose of this third principle is to assure the construction of a program, the execution of which progresses from the beginning to the end, stepwise, in a forward direction.

One of the interesting aspects of the simulation as a test problem is the necessity for the use of a repetition statement to express the first decomposition of the problem. As has been pointed out earlier in this report, the use of the simulator decomposes into a number of independent runs of the simulation program. This requirement is met with what is essentially a "one line program", PDP11SIM,

repeat begin ... end until Stop;

The repeated statement is decomposed until the setting of the control variable is specified.

Perhaps the best example of the net effect of a combination of selection statements is found in the program FINDONEUP (Figure 3.11). This program consists of a simple sequence of two operations:

- 1) isolate the high character of the source field; and
- 2) assign J a value of one for all single operand instructions.

Each of these events can be validated as stand-alone events which must take place in the stated order. If the value of Hisource is 1, 2, 3 or 7, then the instruction cannot be a single operand instruction. If it is 5, then it is a single operand instruction without any question. The other cases, 0, 4 and 6, all require further examination before determining whether or not they are single operand instructions on the PDP-11 computer. The inner structure of each of these further tests is built in the same way.

Because of the structured use of the nested selection statements, the complex selection and the resultant instruction execution is treated as a single computation in a sequence of computations.

Thus, even when the non-sequential forms such as a repetition or a selection statement are used, each is incorporated into the program in terms of its net effect, in a sequential construct.

An illustration of the operation of the fourth principle of Structured Programming can be seen by examining the programs given in this section in their order of occurrence.

In the first program, given in Figure 3.1 (PDP11SIM), the control structure for the program is the repeat statement with the compound statement that it controls. The requirements to complete the control structure are satisfied by the declaration of Stop, the control variable, and by the indication of a component of the program QUERYUSER to provide a value for the variable. The subtasks which must be performed to do the simulation are identified, and any conditions which apply to their execution are specified.

The program RUNSIM again employs program calls or stubs to defer details about the computation which are not required for the complete specification of the control structure at this level. Sufficient details of the program are developed to show how the control variable, Enable, receives its values. The assignment of values to Go is also identified; thus Go is initialized to a true state and can be set to false either as a result of console action or if the cyclecount exceeds the cycle limit. All other detailed specification of the solution is deferred to lower levels of the program organization.

The program INSTRUCTNHNDLR introduces only enough detail into the program to specify the five subtasks into which instruction processing decomposes. The program stubs are used to separate the details involved in each phase of instruction processing from the sequencing controls used to activate the phases. The program composed through this third level of program organization contains none of the details that are directly associated with a PDP-11. The simulator is programmed or specified into the very "heart" of the instruction cycle with only the control structure developed in detail. When the stubs for INSTRUCTNHNDLR are defined (FINDFORMAT, Figure 3.7), then the details of the computer being simulated enter into the program design.

The application of Principle #5, the deferral of decisions about data representation, is well illustrated in the example. In the sample of programs presented there is no discussion of data representation except for the comment contained in

the program INSTRUCTNHNDLR. All requirements to deal with the internal representation of the data in the pseudo memory are isolated to a single level of programs at a very low level. Any concern as to whether the simulator is to be executed on a decimal or a binary machine are isolated from the program structure. The impact upon the program of any particular optimum word width for the host machine is also isolated from all except the lowest levels of the program organization.

5. ANALYZING PROGRAMS FOR THEIR CORRECTNESS

The programs presented in this section are all formed by the decomposition of a programming problem. These programs are constructed in one of three forms. The simplest form of these programs is an ordered set of smaller programming problems which are performed in the specified order (e.g., INSTRUCTNHNDLR, Figure 3.5). Another form of the programs involves the repetition of a smaller programming problem (e.g., PDP11SIM, Figure 3.1). The third case involves the selection between alternatives (e.g., IEXECUTE, Figure 3.9). Thus each of the basic control structures used in Structured Programming can itself be the basis of organization for a program. In each case, the net effect of the performance of the program must be to provide a solution to the programming problem on which it is based. Each program represents the specification of a problem solution.

Since the programs take on a restricted set of forms, some of the requirements for their validation can be generalized. This generalization is a guide for later application to the individual programs.

a. Case I. Sequential Programs

For purposes of analysis certain properties of a well structured sequential program can be taken to be axiomatic.

1. All of the statements in a sequential program will be evaluated once and only once, in the order of their occurrence.
2. All programs have a single entry and exit point and therefore execute from beginning to end.
3. No program can have its local variables altered as a side effect of the execution of some other programs.
4. Only an own variable can retain a state from a previous execution of the program.

The essential problem in the validation of a sequential program can be summarized in questions.

1. Do the statements constitute a decomposition of the problem; i.e., is the problem to be programmed, at a given level, completely satisfied provided the subtasks which occur as calls (procedures, sub-routines or macros) in the program are correctly defined and the control statements (repetition or selection) together with the assignment statements properly executed? This requires consideration of the programming environment at the level being examined as well as the behavior of the PDP-11.
2. Does the order of the subtasks (statements) create the desired effect?

b. Case II. Iterative Programs

Iterative programs present a different problem in validation. The first question from sequential programs (Case I) must be addressed for iterative programs also. In addition to that question, it is necessary to validate the iteration process itself. Thus there are three additional questions to be examined.

1. Is the controlled statement executed at least once when appropriate?
2. Is the minimal requirement for the termination of an iteration statement satisfied? Since side effects are not allowed, the statement, S, must modify the value of one or more variables in such a way that after a finite number of iterations the condition for repetition of S is no longer satisfied.
3. Does the termination take place correctly; i.e., are the proper number of iterations carried out?

c. Case III. Selective Programs

When a program is organized around a selection statement (the selection can be carried out through a case statement or an if statement), there are special criteria for correctness.

1. The number of statements provided as alternatives must exhaust all possible values of the selection variable.
2. The alternative statements must themselves be valid representatives of one of the three forms (usually a sequential form).
3. It must be possible for the selection variable to take on the full range of values.
4. If the only assignment of values to the selection variable is after the selection, then this control variable must be declared at a higher level than the selection statement, and the variable must be initialized.

The conditions presented for the given program form, when satisfied, are a basis for an analytical validation of the program and are conditions which are necessary although possibly not sufficient for a demonstration of the program's correctness.

PDP11SIM is an iterative program in which the statement being controlled is a compound statement. The first step in the validation of the program is an analysis of the control structure of the compound statement being controlled. It starts with a statement of the net effect of the ordered execution of the two procedures, SIMPROG and QUERYUSER. The net effect states: "Carry out a single simulation run and then ask the user if another run is desired." A single simulation run may include the execution of a number of programs for the target machine, the PDP-11. The order of execution for the two procedures is really immaterial. As positioned, the details of the definition of QUERYUSER must correctly support the position adopted relative to SIMPROG.

The validity of the unlabeled compound statement can be asserted since it meets the needs of the problem. There is an implied reservation, however, based upon the valid definition of the procedures SIMPROG and QUERYUSER.

If the net effect of the iteration statement performs multiple executions of the simulation program based upon the user's requirements, then the program is valid. For this effect to be achieved, the compound statement must do exactly what it was specified to do. The form used for the iteration statement guarantees that that there will be at least one execution of the simulation. Within the compound

statement the procedure QUERYUSER returns the value of the control variable, true or false, which reflects the user's needs. Therefore, the iteration statement can terminate. If $\text{Stop} = \text{false}$ on the n th iteration, then there will be no iteration $n + 1$. All of the conditions for a valid iteration program are met.

SIMPROG is a sequential program defined within PDP11SIM. There are two (2) specific requirements for the validation of a sequential program. If the program can be shown to have as its net effect a single simulation run, then it is valid. The three procedures INITSIM, RUNSIM and CLEANUPSIM, if properly defined, are a decomposition of the functions of a simulation run and they are expressed in the only order that is appropriate to performing the task. An integer variable is provided to communicate a user-supplied cycle limit value from INITSIM to RUNSIM where it is used to assure termination.

The program SIMPROG can be asserted to be valid.

RUNSIM (Figure 3.3) consists of the repetition of a compound statement. The desired net effect to be achieved by RUNSIM is the performance of a single simulation run. A single simulation run consists of cycling to carry out the simulation of either instruction processing or console actions as long as required. The single statement being repeated in the program, RUNSIM, will continue until the value of Go is false. With each cycle of the iteration the program will execute either of two programs, INSTRUCT or CONSOLE, to perform the required instruction processing or console actions. The choice of which program to execute is based on the value of the variable, Enable.

Enable is initialized to the false state, which causes the program CONSOLE to be selected on start up. Within this program the variable Enable can be set to true; since it can be reset to false within the program INSTRUCT, it is clear that it is possible for Enable to take on the full range of values. The conditions under which the variable is set must match those found in the PDP-11. Enable is assigned the value true only if Halt is false and either Continue or Start is true. Continue and Start are reset to false each time the program CONSOLE is entered. Unless the Halt variable is previously set to false before Continue or Start is reset, instruction processing will not commence since Enable would not be set to true. The net effect of this combination of switch actions is the same as that found in the PDP-11.

Within the program INSTRUCT the variable Enable is assigned the value false if a variable Haltinst is true. Haltinst is initialized to false with each iteration of INSTRUCT. It is passed as a parameter to the program INSTRUCTNHNDLR. If the program decodes a Halt command, then it is expected that the value true will be assigned to Haltinst; this then allows Enable to be assigned the value of false. Thus, a halt command stops the iteration of instruction processing and initiates CONSOLE to permit simulated console actions.

It can be concluded then that the compound statement controlled by the repetition does perform the proper functions as required by the problem.

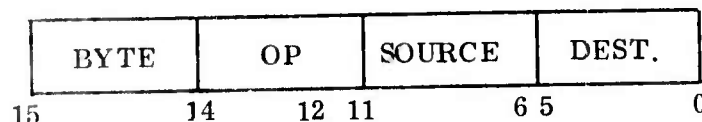
The variable Go can be set to false by a simulated console action. This is an extension of the PDP-11 which is equivalent to turning off the power to the computer. The variable Go is also set to false as a result of a limit placed on the allowable number of iterations of the instruction processing.

It should be noted at this level that one facility of the PDP-11 console is lacking in the model - allowing the user to set a simulated halt switch while Enable is set to true and PDP-11 instructions are being interpreted.

INSTRUCTNHNDLR (Figure 3.5) is essentially a sequential program formed by identifying an ordered set of subtasks which completely perform the task of instruction processing. ISELECT is assigned the task of guiding a re-direction of the instruction stream due to an interrupt, if necessary. The responsibility to cause the word pointed to by the program counter (PC), register 7 (R7), to be brought to a pseudo instruction register, Instruction, is given to IFETCH. A partial decoding of the instruction to identify the number of operands required for execution is the task of FINDFORMAT. Based upon the necessity for an operand as determined by FINDFORMAT, the task OPFETCH may be selected. If selected, OPFETCH retrieves the required source and/or destination data. IEXECUTE must complete the decoding of the instruction, select the proper function, and execute it, as well as store the result if required. When executed in this order, the procedures identified, if properly defined, will effect an instruction processing cycle as defined for the PDP-11.

INSTRUCTNHNDLR can be considered a valid program.

The program FINDFORMAT (Figure 3.7) consists of three subtasks. The instruction format which is the basis of this syntactic analysis program is the double operand format.



Double Operand Format

The three subtasks are to extract the byte indicator, copy the op field, and then analyze the op field. Both the order and the nature of the subtasks satisfy the assigned programming problem. The third subtask, however, is a complex selective

statement, the net effect of which must be confirmed. There are four (4) sub-classes of requirements identified by the contents of the op fields dealt with in the compound statement OPFIELD. The four sub-classes, their identification criteria, meaning, and implied action are summarized in Table 3.1 below.

	<u>OP</u>	<u>MEANING</u>	<u>ACTION</u>
(1)	0	Not double operand.	Examine source field.
(2)	1 - 5	Double operand; Byte operations allowed.	J = 2, byte indicator valid.
(3)	6	Double operand; Addition or subtraction; No byte operations	J = 2, byte indicator distinguishes addition from subtraction, save it. Set byte to <u>false</u> .
(4)	7	Illegal operation.	No operands. J = 0.

Table 3.1. Opfield Contents

The compound statement OPFIELD opens with a selection statement of the form:

if (Boolean = true), then S₁ else S₂ .

This binary selection is based on the zero and non-zero classes in the opfield. In the zero case, control is passed to FINDONEOP, with J = 0 as a result of initialization. If Op is non-zero, then the statement TRY2 is executed. If OP = 7, then an exit is made from TRY2. Control is returned to the outer block of the program with J = 0. Since the instruction is illegal there is no operand required. If OP ≠ 7 and OP ≠ 0, then the instruction is legal; it remains to be determined if it is an add or subtract. In any case, J is assigned the value 2, to indicate that two operands are required. The isolation of the add and subtract instruction is accomplished by the selection based on (OP = 6).

Table 3.2 relates J values to all of the possible bit combinations for an instruction. Based upon this and the analysis it appears that FINDFORMAT is a valid program.

J	Op	Source	Dest.		Op	Source	Dest.	Instruction Class
0	17	00	00	$\leq I \leq$	17	77	77	Illegal
2	16	00	00	$\leq I \leq$	16	77	77	Subtract Instructions
2	11	00	00	$\leq I \leq$	15	77	77	Double Operand - Byte Instruction
0	10	64	00	$\leq I \leq$	10	77	77	Illegal
1	10	50	00	$\leq I \leq$	10	63	77	Single Operand - Byte Instruction
0	10	44	00	$\leq I \leq$	10	47	77	Trap Instruction
0	10	40	00	$\leq I \leq$	10	43	77	EMT Instruction
0	10	00	00	$\leq I \leq$	10	37	77	Conditional Branch Instruction
0	07	00	00	$\leq I \leq$	07	77	77	Illegal
2	01	00	00	$\leq I \leq$	06	77	77	Double Operand Word Instruction
0	00	64	00	$\leq I \leq$	00	77	77	Illegal
1	00	50	00	$\leq I \leq$	00	63	77	Single Operand Word Instruction
1	00	40	00	$\leq I \leq$	00	47	77	JSR *
0	00	04	00	$\leq I \leq$	00	37	77	Conditional Branches
1	00	03	00	$\leq I \leq$	00	03	77	SWAP BYTES
0	00	02	40	$\leq I \leq$	00	02	77	Condition Codes Operators
0	00	02	10	$\leq I \leq$	00	02	37	Unspecified
1	00	02	00	$\leq I \leq$	00	02	07	RTS
1	00	01	00	$\leq I \leq$	00	01	77	JUMP *
0	00	00	06	$\leq I \leq$	00	00	77	Illegal
0	00	00	00	$\leq I \leq$	00	00	05	Operate Instruction

I stands for Instruction.

*Special check in instruction execution to be sure the destination is not a register.

Table 3.2. J Assignments All Possible Instruction Patterns

FINDONEOP (Figure 3.11) is a selection program (Case III). The form taken by the selection statement is the case statement

"case i of { $S_0, S_1, S_2, S_3, \dots, S_n$ } when $0 \leq i \leq n$."

Confirming the general criteria for validation shows that the control variable Hisource has a range from 0 through 7. Since the statement list contains eight (8) entries, all possible values of the control variable are accounted for. Hisource takes its value from bits 11 through 9 of the instruction being processed and thus may take any of the values from 0 through 7. The control variable is local to FINDONEOP; thus there are no scope problems. A new value of Hisource is derived for each execution of FINDONEOP. The remaining general requirement for correctness of FINDONEOP involves validating the alternatives in the statement list.

Entry S_0 , SOURCELOW, a procedure declared as a part of FINDONEOP, is itself a selection program. The control variable for this second order selection is Losource. It is also derived from the source field of the instruction local to FINDONEOP, and can take on the value of the eight (8) integers from 0 through 7. The statement list has eight (8) entries. The entries S_0, S_4, S_5, S_6, S_7 are vacuous because all instructions which decode these values in bits 8, 7 and 6 are instructions which have no operands (Operate Group and Conditional Branches). The statement S_1 decodes to a Jump command with a destination operand, so J is assigned a value of 1. Statement S_2 from the list is a procedure named TRYDES, a sequential program which correctly differentiates between an RTS instruction and a Condition Code Operator. If the instruction is recognized as an RTS, then J is assigned a value of 1. The statement S_3 assigns J the value of 1 because the instruction is recognized as a swap byte command. Thus, all non-vacuous entries in the statement list are valid, which leads to an assertion that SOURCELOW is itself valid.

Returning to the original solution statement, S_1, S_2 and S_3 are vacuous and S_4 leads to a procedure which differentiates between the JSR command and the Trap command; S_5 assigns J a value of 1 since all instructions in that class are members of the single operand group. S_6 is defined by a procedure which indicates the four shift commands and sets J equal to 1 for these values. All others in the S_6 group are left at $J = 0$. The S_7 condition defaults $J = 0$ through the vacuous entry.

The set of PDP-11 instructions which contain their distinguishing op codes in the source field are processed to identify those which require a single operand. All possible bit combinations in the source field are processed and handled. A summary of the information encoded in the source field and its significance as an operator indication is shown in Table 3.3. Based on the satisfaction of the general

criteria for selection programs, and the relationship to the PDP-11 instructions, it is asserted that the set of all possible bit combinations in the source field are properly identified as belonging to one of two subsets, the one requiring a single operand for execution, the other requiring none. FINEONEOP appears to be valid.

<u>2 Digit Source Field</u>	<u>Meaning</u>	<u>Action</u>
64 - 77	Illegal	J = 0
50 - 6	Single Operand	J = 1
40 - 47 $\wedge \neg$ Byte	JSR	J = 1
40 - 47 $\wedge$ Byte	TRAPS	J = 0
4 - 37	Conditional TRANS	J = 0
1 - 3 $\wedge$ Byte	Conditional TRANS	J = 0
3 - 3 $\wedge \neg$ Byte	SWAP BYTE	J = 1
2 - 2 $\wedge \neg$ Byte Dest. < 10	Condition Code Operator	J = 0
1 - 1 $\wedge \neg$ Byte	JMP	J = 1
0 - 0 $\wedge \neg$ Byte		J = 0

Table 3.3. Source Field as Operation Indicator

OPNONE (Figure 3.14) is a selection program. The program is built of nested binary selections. Due to the nesting of the statements a table (Table 3.4) of all possible bit combinations for the instructions processed by OPNONE helps in the validation process. The control variables used in the selection statements are all booleans. For ease of discussion the nested levels are identified as Zero (0) through Six (6). The zero level is an implied identification. The use of the nested

"if B then S_1 else S_2 ;

form of selection divides the list pattern into progressively smaller subsets. Table 3.5 indicates the bit patterns that are isolated and the levels at which they are isolated. Based on the tables, it can be asserted that OPNONE is a valid program.

An argument has been presented to demonstrate the validity of each of the programs used in this Section of the report.

Range				Identification			Level Handled	
17	00	00	$\leq I \leq$	17	77	77	Illegal	0
10	64	00	$\leq I \leq$	10	77	77	Illegal	0
10	44	00	$\leq I \leq$	10	47	77	Trap	1
10	40	00	$\leq I \leq$	10	43	77	EMT	2
10	00	00	$\leq I \leq$	10	37	77	Conditional Branch	3/4
07	00	00	$\leq I \leq$	07	77	77	Illegal	0
00	64	00	$\leq I \leq$	00	77	77	Illegal	0
00	04	00	$\leq I \leq$	00	37	77	Conditional Branch	3
00	02	40	$\leq I \leq$	00	02	77	Condition Codes Operators	5
00	02	10	$\leq I \leq$	00	02	37	Unspecified - Illegal	6
00	00	06	$\leq I \leq$	00	00	77	Illegal	6
00	00	00	$\leq I \leq$	00	00	05	Operate Group	6

Table 3.4. Possible Instruction Bit Patterns at OPNONE (J = 0)

Level/ Statement	Where Directed							Trap No.
0/S1	17	00	00	-	17	77	77	10
	10	64	00	-	10	77	77	
	07	00	00	-	07	77	77	
	00	64	00	-	00	77	77	
1/S1	10	44	00	-	10	47	77	34
2/S1	10	40	00	-	10	43	77	30
3/S1	10	04	00	-	10	37	77	CONDBRNCH
	00	04	00	-	10	37	77	
4/S1	10	00	00	-	10	03	77	CONDBRNCH
5/S1	00	02	40	-	00	02	77	CONDITION
6/S1	00	02	10	-	00	02	37	10
	00	00	06	-				

SECTION IV

THE DIJKSTRA PROGRAMS AND THIS STUDY

Before proceeding to a review of the results of this study effort, it is necessary to summarize the material presented by Dijkstra.² Within the Dijkstra paper there are two programs presented in detail - "print the first thousand prime numbers" and "plot $y = f(x)$ " - as examples of the composition of structured programs. They differ significantly in form, and these differences are sufficiently great that their common reliance on a set of axioms about programs and their formation is obscured. These axioms are the foundation for the principles of Structured Programming.

Axiom 1. All programming problems can be parsed, with respect to time, into a set of sequential sub-actions.

This parsing is the very basis of computers and programs. The parsing consists of the division of a problem into an ordered set of smaller subtasks. In the prime number example²¹ the task:

"print the first thousand prime numbers"

is given. This task is then parsed into the two sub-actions:

"fill table p with first thousand prime numbers"

"print table p."

The decomposition of some tasks leads to a repetition of the same sub-action. An example of repetition is shown in the parse of the second statement from the example aforementioned:

"print p [k] for k from 1 through 1,000."

To be computable the number of repetitions of a subtask must be finite, but this number can be indeterminant, as in an iteration. A shorthand representation of this special form of parsing is made through the use of the repetition statement:

"repeat S, until C."

In this form the single statement which is controlled represents a parse with respect to time into an ordered set of sub-actions.

Axiom 2. Programs may be considered to be computationally equivalent if they evoke computations which have the same net effect.

The net effect of a program is defined as the net change which has taken place across an interval of time between the times t_0 and t_1 . The time of the beginning of the computation is denoted by t_0 , while t_1 identifies the terminal point of the execution. Such an interval of time is considered as an entity. Any intermediate changes in the state of the computation process are ignored. If there are no possibilities of side effects from other computations, those changes seen in the process state are the net effect of these computations.

Corollary: For net effect comparisons to be practical in large programs, it is often necessary to be able to map the sequence of one program upon the other.

Such a mapping may often require the re-ordering of the sequence of one of the programs. The sequence of a program can be re-ordered by changing the order of the sub-actions if there is no change in the net effect of the program. Sometimes comparisons can only be made at a more abstract level of the programs. This more abstract state can be reached by applying the inverse of the operation described in Axiom 1.

Axiom 3: Any problem statement with its inputs and outputs identified can be considered a detailed program for an abstract "special purpose" machine.

Given a machine the only purpose of which is to print a list of prime numbers, a complete program for this machine is expressed in a single statement:

begin print first thousand prime numbers; end;

A machine of this kind generally exists only as an abstraction. Writing a program can be described as the process of reducing a problem to the logic and control required to transform a more general purpose machine into such a special purpose machine.

Corollary: A problem statement can be considered to be an abstract program for a general purpose machine.

The program is abstract because it deals with the solution of the problem in general terms rather than in the particular instructions of the general purpose computer.

These philosophical points are the basis of Dijkstra's efforts to build programs so that their texts will support a convincing demonstration of correctness. Yet the application of the axioms and their corollaries in the two examples given by Dijkstra ("list 1,000 prime numbers" and "plot the graph $y = f(x)$ ") seem to yield radically different program forms, even though the essentials of the programs are the same. Neither of the examples results in a program that can be executed, except in the environment of special computer systems.

Both of the programs composed in the Dijkstra paper present a progression of programs. Each of the programs taken in order in the progression involves more detail than its predecessor. This gradual refinement and functional parsing of the total task acts as a framework on which the analysis necessary to demonstrate correctness can be based.

The examples presented by Dijkstra are relatively short programs in contrast to the problems of the real world. Both examples parse immediately into at least two sequential actions. In fact, they are representative of a whole class of typical programs involving a two step process:

- 1) derive a data set;
- 2) display the result.

The first example chosen by Dijkstra,²² hereafter referred to as Case I, reports the step-wise development of a program to print the first 1,000 prime numbers. This program does not assume the algorithms for the "sieve of Eratosthenes", but rather develops its own algorithm through a gradual process of refinements. These refinements make use of the mathematical properties of prime numbers to produce a reasonable computational algorithm, which is a variation of the "sieve of Eratosthenes."

The second example, Case II,²³ involves the printing of a graph upon a line printer. The printer is given to have only two commands, "New Line Carriage Return" ("NLCR") and "Print Symbol (n)" ("PRYSM(N)"). NLCR

defines the left-most position of the next line as the "currently printable position;" PRYSM(N) prints a character identified by the value of the integer parameter N on the currently printable position, and defines the next position as the new currently printable position. The problem to be solved is to plot the form given in a discrete parameter representation upon the digital printer.

The form of Case I, the prime number example, is illustrated by the three levels of description given in Figure 4.1.

description 0:	
<u>begin</u> "print first thousand prime numbers" <u>end</u>	
description 1:	
<u>begin</u> variable "table p";	1a
"fill table p with first thousand prime numbers";	1b
"print table p";	1c
description 2:	
1a = " <u>integer array</u> p 1:1000"	2a
1b = "make for k from 1 through 1000 p [k] equal to kth prime number"	2b
1c = "print p [k] for k from 1 through 1000"	2c

Figure 4.1. Dijkstra's Prime Number

This program, Case I, can be characterized as follows:

1. Makes explicit use of the problem statement to form the initial effort at a program.
2. Step-wise development of levels are based upon the analysis of the program and directly reflect the analysis. In each, the machine for which the proposals are to be considered as programs remains an abstract concept. This machine exists outside the abstract state only when the syntax in which the problem is described maps into the syntax of some existent computer system.

3. Each level is presented as a structure which approaches a program or procedure in form, but in an informal way. The term proposal is used to describe these entities.
4. The proposals themselves are formulated as a sequence of steps expressed in an informal syntactic way.
5. Proposals are identified through the use of a functionally non-descriptive identifier. The identifiers are used to describe the historical position of the proposal as a derivative of a predecessor. Where a proposal is presented to refine a particular statement in a predecessor, this derivative relationship is exhibited in the identifier for the proposal. Alternative refinements to the same predecessor have the same root identifier augmented with markers. Multiple alternatives can be developed for the same proposal level.

Thus, version 1 of the refinement of line 2b is developed and is labeled 2b1(1).

description 2b1(1):

```
begin p [1] := 2; p [2] := 3; p [3] := 5; p [4] := 7;
      p [5] := 11; ... end
```

A version 2 of this description recognizes that the programmer does not know the prime numbers and must derive them.

description 2b1(2):

```
begin integer k, j; b := 0; j := 1;
      while k < 1000 do begin "increase j until next prime
      number";
      k := k + 1; p [k] := j end
end
```

The program development illustrated by Case II, on the other hand, offers a contrast in each of these areas.

The first levels of text proposed for the graph problem illustrate the accompanying comments.

COMPFIRST

begin

draw: {build; print } ;

var image;

instr build (image), print (image)

end

where,

var is an identifier for a variable list,

instr is an identifier for an instruction list for the machine COMPFIRST.

The second text proposed refines the instruction build.

CLEARFIRST

begin

build: {clear; set marks} ;

instr clear (image), set marks (image)

end

The characteristics of Case II are:

1. The use of the problem statement is implicit.
2. Development in the second example involves the explicit use of the machines as a level of development. Each machine is formed to be capable of explicitly executing an algorithm. This algorithm is included in the machine definition. The state space of the machine is defined, as well as the instruction set required for the execution of the algorithm. The entire package is called a pearl and is given the name of the machine. The analysis for the problem in Case II takes place behind the

scenes. This analysis is represented in the machine manuals, which Dijkstra suggests be developed for the machines. In the example given, the meaning given to the instructions looks much like the proposal steps given in Case I.

3. The programs which are written for the machines as they are presented in Case II are formulated in a clear functional notation. Thus, a program can be illustrated with the program from COMP-FIRST

draw: {build; print}; .

In this notation what might be considered a level 0 definition for Case I consists of a sequence of two steps. This program is executed by calling two procedures in order. In this way the developmental notion is much closer to an executable form than is found in Case I.

4. All proposals are presented as programs, together with the machine defined for them. There can be more than one program written for a single machine. The programs are related to earlier proposals through their identifier. All program identifiers except the first program identifier represent undefined instructions for earlier machines. The machine specified for "draw", the first proposal, is named COMPFIRST. It has a state space "image" and two undefined instructions, "build" and "print".
5. Identifiers exist in Case II in two levels, machine identification and program identification. In both cases the identifiers used are descriptive of the function associated with the machine or program, respectively. The machine identifier COMPFIRST tells that computation precedes display. The program identifier "draw" reflects the function to be performed by the program. Thus the machine name reflects the ordering of the sub-actions and the name "draw" specifies the net effect of their execution.

6. The levels of machines in Case II are implicit in that a single level of a machine can be programmed in terms of lower level machines only. In turn, the functions which are refined by the programs written for that machine are available to higher level machines. Thus the relationship between the various levels of the design can only be noted by referencing the machines themselves.

A close examination shows that the examples present the same material in different ways; these differences are a reflection of a change in emphasis from design analysis to a closer approximation of the execution form. Neither of these forms of step-wise program composition yields what could be considered a compilable or executable form.

As the material of Case I is developed by Dijkstra, the procedures which are executable still must be connected. Thus, when the decomposition of the statements 2b and 2c are completed, a separate program must be built of the derived parts which will activate them as elements in the sequence. Although the format of Case II does eliminate this problem, the executable programs are buried in the middle of the machine definitions. Some researchers have designed systems which would execute the pearl type of structure used in Case II.²⁴ The procedure identifier together with the machine name form a name-couple which could serve to uniquely identify them in a computer system. It is necessary to extract the executable procedures from their positions inside the machine definitions.

It should also be recognized that the two techniques presented in the examples have much in common.

1. Both of the examples make use of the technique of step-wise decomposition of the problem through a step-wise composition of a program to solve the problem.
2. The goal of constructing a program is addressed through this process in both examples.
3. Design decisions are deferred in both of the techniques.
4. Neither of the composition processes have hard and fast rules about how long the decisions are delayed.

5. Both approaches generally tend to place priority on the development of the controls for the program. Thus, when a function will be performed is completely defined before the function is defined.
6. Simply ordered processes are the preferred manner of expressing a parse of an action in both of the examples.
7. Introduction of selection and repetition into the control path of the partitioning process is delayed until after the net effect has appeared in a sequential parsing of the process under both approaches.

Using these common features (expressed in Section II as Principles of Structured Programming), the programmer constructs a program of an ordered set of "small" sub-actions to create a desired net effect. The resultant program organization is an aid to the programmer in constructing a convincing demonstration of correctness for the program.

The seven common points observed here were compressed and re-stated to form the principles given in Section II, and it is upon this basis that the work reported in the study was developed. Thus, the work presented in Section III to illustrate the application of the philosophy of Structured Programming has retained the substance of Dijkstra's work while essentially adopting the form of Mills.²⁰

This form is constructed by explicitly using the functional program form of the programs found in the "pearls" of Case II. There does not appear to be any advantage in building an explicit form of machine as found in the "pearl". The analytical development of a problem, found in the Case I example, has been relegated for use in the designer's notebooks, at least until the programmer's analytical insights become keen enough to overcome the drawbacks discovered during the course of this study.

SECTION V

RESULTS AND OBSERVATIONS

The results of the study consist of a statement of a set of Principles of Structured Programming and a program built using these principles. A further result of this study effort is a pair of guides for limiting the size of the program produced at each level. These guides are subjective as contrasted with Mills' "one page limit".¹⁹ It is evident from the experiments conducted in this study that the complexity of the statements used in a program should limit the size of the program. It is reasonable to build a convincing demonstration of correctness for a program consisting of 50 sequential subtasks; it may not be reasonable to build a similar demonstration for a program consisting of a page full of nested selection statements. Thus the ability to be convincing about the correctness of a program forms an upper bounds upon the size of any particular program construct. A lower bound on the minimal size of a program is a function of the controls required in the program. The details of the decomposition of an action must be sufficient to indicate the setting of the control variables used in the program. Any conflict arising between the limits can be resolved by using an alternate form of decomposition.

The programs given in Section III contain an illustration of the resolution of such a conflict. It was necessary to split the FINDFORMAT program into the two programs, FINDFORMAT and FINDONEOP, because of the complexity of the control path. The conflict which arose in FINDFORMAT was resolved by re-analyzing the task until sufficient understanding was achieved to allow the use of case statements as a selection mechanism. It has been found that when the upper and lower bounds criteria are applied to a program, it has been possible to quickly construct a flow diagram for the program.

A guide to the demonstration of the validity of the programs is included in Section III, together with an example of its application. This guide is a by-product of the effort, and while not sufficient for a proof of correctness, it has been very helpful in desk checking the programs and in building arguments for validity.

1. OBSERVATION

The study clearly revealed the elusive nature of the concepts of Structured Programming. The prior lack of a clear presentation of the principles of Structured Programming was recognized during the investigation; as a result, a set of principles has been enunciated in this report. These principles can serve to guide further research and development efforts. They also can be applied to avionics systems currently in development and procurement.

While the study exposed ambiguities existent in the concepts of nested levels of abstraction, the concepts are not made inoperative by these ambiguities. Rather, the study finds merit in the notion and a need for further clarity beyond that developed in the course of the study.

Many researchers see this as an inter-relationship of all sub-actions of tasks which deal with a comparable level of detail. This interpretation appears to be consistent with the reports Dijkstra made of "THE" system.¹⁵ However, the fact must be observed that each of these levels may contain a number of layers. Further, there are no rules for use as guides in allocating programs to levels. This is very much a function of the insights of the programmer.

2. FAMILIES OF PROGRAMS

Structured Programming appears to relate to the generation of families of programs using one member of the family as a base. This indicates a potential impact on other software research and development work currently in progress, particularly the AED work of Ross.²⁶

3. PROGRAM CRITICAL PATHS

There is a practical interest in recognizing the critical paths in aerospace software systems. Such a recognition often can lead to the removal of processing bottlenecks. When the relative frequency of execution of the sub-components of a program are known, then they can be intelligently evaluated as potential sources of bottlenecks in the system. The use of a top-down program organization exposes hierarchical ordering of the elements of the program, which facilitates the assignment of relative frequency of execution values by analytical means.

In the test problem, the program path from INSTRUCTNHNDLR down represents the principal computation path. One of its subtasks is not executed as frequently as the others. (The operand fetch subtask is not executed for instructions having a j value of zero (0).) A frequency distribution of the instructions used in typical programs serves to provide a weighting value for this subtask.

4. RECOGNIZING COMMON FUNCTIONS

In applying the principles of Structured Programming, control for a function is developed before the function is defined. The limited experience gained in this

test program, supported by previous experience²⁷ indicates that this approach leads to a greater recognition of common sub-functions. An example of this is seen when the instruction fetch is recognized as being identical to the stack pop operation used in the operand fetch. Thus, the frequency of use of this operation is obtained by adding the two uses of this common function. Although there is insufficient evidence to evaluate just how much the organization of program structure aids in the recognition of these common functions, it would appear to be significant.

5. ERROR DETECTION IN PROGRAMS

In preparing the demonstrations of program correctness, a number of errors were found. Other errors were avoided simply because of the orderly way in which the target program was organized. Although the reliability of software is a function of the individual programmer as well as his environment, there are many factors in the use of well-structured programs which appear to contribute to reliability in the software. From this experience as well as the support of other experiments²⁵ there is reason to believe that the use of Structured Programming contributes significantly to the reduction of programming errors.

6. MEMORY EFFICIENCY

The impact of Structured Programming efforts upon memory efficiency is basically unpredictable. Any impact which results from Structured Programming is a function of other factors, such as the size of the system. The larger a system is, the greater the probability that there are common sub-routines. Structured Programming generally leads to much smaller sub-programs than are currently found in avionics systems. There are preliminary indications from other studies that the frequency of utilization of a sub-program tends to vary inversely with its size. This would suggest that Structured Programming would lead to a set of frequently used sub-programs.

In order for the maximum benefit to be derived from the use of common sub-routines, it is necessary for the system to allow shared code. Such a system would allow a single copy of the instructions to serve all users.

Any advantages which accrue in memory utilization may be offset by requirements for control of sub-routine communication.

7. **HARDWARE - SOFTWARE TRADE-OFFS**

There is insufficient evidence in this study to draw any conclusions with regard to hardware-software trade-offs. The Structured Programming approach does allow for the replacement of software functions by hardware with a minimal impact upon the program.

REFERENCES

1. Dijkstra, E.W. "GOTO Statement Considered Harmful," Communications of the ACM, vol. 11 (March 1968), pp. 147-148.
2. Dijkstra, E.W. "Notes on Structured Programming," Technische Hogeschool Eindhoven, Report No. EW-D-249, 70-WSK-03, 2nd edition (April 1970).
3. Dijkstra, E.W. "Notes on Structured Programming," Structured Programming, (Dahl and Dijkstra, eds.), Academic Press, New York (1973).
4. Bohm, C. and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of the ACM (May 1966), pp. 366-371.
5. Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," Communications of the ACM, vol. 12 (October 1969), pp. 576-580.
6. Mills, H.D. "Mathematical Foundations for Structural Programming," IBM Corporation, Gaithersburg, Maryland, FSC 72-6012 (February 1972).
7. Rice, J.R. "The GOTO Statement Reconsidered," Communications of the ACM, vol. 11 (August 1968), p. 538.
8. Dijkstra, E.W. "A Reply by E.W. Dijkstra (to John R. Rice)," Communications of the ACM, vol. 11 (August 1968), pp. 538-541.
9. Leavenworth, B.M. "Programming With(out) the GOTO," Proceedings of the ACM 25, vol. II (1972), ACM, New York, pp. 782-786.
10. Hopkins, M.A. "In Support of GOTO," Proceedings of the ACM 25, vol. II (1972), ACM, New York, pp. 787-790.
11. Wulf, M.A. "A Case Against the GOTO," Proceedings of the ACM 25, vol. II (1972), ACM, New York, pp. 791-796.
12. London, R.L. "Correctness of a Compiler for a Lisp Subset," Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7 (January 1972), pp. 121-127.
13. London, R.L. "Proving Programs Correct: Some Techniques and Examples," BIT, vol. 10, no. 2, pp. 168-182.
14. Good, D.I. "Toward A Man Machine System for Proving Program Correctness," (Thesis Ph.D.), University of Wisconsin (1970).

15. London, R.L. "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence 5, American Elsevier Publishing Company, New York (1970), pp. 569-580.
16. Dijkstra, E.W. "The Structure of the T.H.E. Multi-Programming System," Communications of the ACM (May 1968), pp. 341-356.
17. Mills, H. "Top Down Programming in Large Systems," Debugging Techniques in Large Systems, (Rustin, R., ed.), Prentice-Hall (1971).
18. Dijkstra, E.W. "Notes on Structured Programming," Technische Hogeschool Eindhoven, Report No. EW-D-249, 70-WSK-03, 2nd edition (April 1970) (Notes, p. 1).
19. Mills, H. "Structured Programming in Large Systems," unpublished memo (can be obtained from author), p. 10.
20. Digstaff. "PDP-11 Handbook," sec. edit., Digital Equipment Company, Maynard, Massachusetts (1970).
21. Dijkstra, E.W. "Notes on Structured Programming," Technische Hogeschool Eindhoven, Report No. EW-D-249, 70-WSK-03, 2nd edition (April 1970), p. 36.
22. *ibid*, pp. 36-49.
23. *ibid*, pp. 64-75.
24. Snowden, R. "System for the Preparation and Validation of Structured Programs," Program Test Methods (Hetzl, W.C., ed.), Prentice-Hall, Englewood Cliffs, New Jersey (1973), pp. 57-74.
25. Baker, T.F. "System Quality through Structured Programming," Fall Joint Computer Conference (1972).
26. Ross, D.T. "Introduction to Software Engineering," Report ESL-R-405, Electronic Systems Laboratory M.I.T., Cambridge, Massachusetts (1969), p. 223.
27. McGonagle, J.D. "A Study of a Software Development Project, Final Report," Contract No. F04701-71-C-0373, CCIP-85, Air Force Systems Command, Los Angeles (1971).

BIBLIOGRAPHY

Aaron, J. D. "The Super-Programmer Project," Software Engineering Techniques, NATO Science Affairs Division (April 1970), pp. 50-52.

Atwood, J. Wm. "I/O Supervision in the Project Sue Operating System," Computer, vol. 6 (November 1973), pp. 19-23.

Aslanian, R. and Bennett, M. "Evolutive Modelling and Evaluation of Operating and Computer Systems," Research Report CA-016, Compagnie International pour l'Informatique, France.

Baker, F. T. "Chief Programmer Team Management of Production Programming," IBM Systems Journal, vol. 11, no. 1 (1972).

Baker, F. T. "System Quality through Structured Programming," FJCC 1972.

Baker, F. T. and Mills, H. D. "Chief Programmer Teams," Datamation, vol. 19 (December 1973), pp. 58-61.

Bohm, C. and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of the ACM, (May 1966), pp. 366-371.

Burkhardt, W. H. and Randel, R. C. "Design of Operating Systems with Micro-Programmed Implementation," National Technical Information Service PB-224 484/6WC, 1973.

Carey, L. J. "IEEE Symposium on Software Reliability," Datamation, vol. 19 (October 1973), pp. 119-125.

Cheatham, T. E. "The Recent Evolution of Programming Languages," Proceedings of the IFIP Congress 71 (Freeman, C. V., ed.), North-Holland, Amsterdam (1972), pp. 298-313.

Chevy, L. L. "Some Case Studies in Structured Programming," MITRE, MTR 2648 VI, 1973.

Cincinnati, University and Draper Laboratory. "Top Down, Bottom Up Structured Programming and Program Structuring," National Technical Information Service (NTIS) N 73-25211 72.

Clark, R. L. "A Linguistic Contribution to GOTO-less Programming," Datamation, vol. 19 (December 1973), pp. 62-63.

Cole, N. M. and Sukel, M. J. "Solving a Software Design Problem Using Plain English," Datamation, vol. 19 (October 1973), pp. 101-106.

Dijkstra, E.W. "Programming Considered as a Human Activity," Proceedings of the IFIP Congress, 1965, Spartan Books, Washington, D.C. (1965).

Dijkstra, E.W. "The Structure of the 'THE' Multiprogramming System," Communications of the ACM (May 1968), pp. 341-346.

Dijkstra, E.W. "A Constructive Approach to the Problem of Program Correctness," BIT, vol. 6 (1968), pp. 174-186.

Dijkstra, E.W. "GOTO Statement Considered Harmful," Communications of the ACM, vol. 11 (March 1968), pp. 147-148.

Dijkstra, E.W. "Reply to Letter on GOTO Statement," Communications of the ACM, vol. 11 (August 1968), p. 538.

Dijkstra, E.W. "Notes on Structured Programming," EWD 249, Technische Hogeschool Eindhoven (1970).

Dijkstra, E.W. "Hierarchical Ordering of Sequential Processes," EWD 310, Technische Hogeschool Eindhoven.

Dijkstra, E.W. "Structured Programming," Software Engineering Techniques, NATO Science Affairs Division, Brussels 39 (April 1970).

Dijkstra, E.W. "The Humble Programmer," ACM Turing Lecture 1972, EWD 340, Technische Hogeschool Eindhoven (1972).

Donaldson, J.R. "Structured Programming," Datamation, vol. 19 (December 1973), pp. 52-54.

Ershow, A. P. "Aesthetics and the Human Factor in Programming," The Computer Bulletin, vol. 16 (July 1972), pp. 352-355.

Gallaher, L. J. "Letter on the Impact of GOTO-less Programming," The Computer Journal, vol. 16 (August 1973), pp. 284-285.

Harris, L. R. "Logical Control Structure for APL," National Technical Information Service AD 766 542/5WC (1973).

Henderson, P. and Snowden, R. "An Experiment in Structured Programming," BIT 12 (1972), pp. 38-53.

Hansen, P.B. "Concurrent Programming Concepts," ACM Computing Surveys, vol. 5, no. 4 (1973), pp. 223-245.

Hansen, P.B. "Structured Multiprogramming," Communications of the ACM, vol. 15 (July 1972), pp. 574-578.

Hoare, C.A.R. "Proof of Programs, Partition and Find," The Queens University of Belfast, Department of Computer Science Report, 1969.

Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," Communications of the ACM, vol. 12 (October 1969), pp. 576-580.

Hoare, C.A.R. "A Structured Programming System," The Computer Journal, vol. 16, no. 3 (August 1973), pp. 209-215.

Hoare, C.A.R. "Proof of a Program Find," Communications of the ACM, vol. 14 (January 1971), pp. 30-45.

Hopkins, M.A. "In Support of GOTO," Proceedings of the ACM 25, vol. II (1972), ACM, New York, pp. 787-790.

Horning, J.J. and Randell, B. "Process Structuring," ACM Computing Surveys, vol. 5, no. 1 (March 1973), pp. 5-29.

Hull, T.E., Enright, W.H. and Sedgwick, A.E. "The Correctness of Numerical Algorithms," Proceedings of the Conference on Proving Assertions About Programs (January 1972), pp. 66-73.

Irmscher, M. "Structure of a Program Documentation Represented by an Example of the Documentation of an Operating System," Rechentich Datenverorb (Germany), vol. 10, no. 7 (July 1973), pp. 13-15.

Iverson, K. "Programming Notation in Systems Design," IBM Systems Journal (June 1963), pp. 117-128.

Jackson, M. and Sanwick, A.B. "Segmented-Level Programming," Computers and Automation, vol. 18 (February 1969), pp. 23-26.

Jones, C.B. "A Formal Development of Correct Algorithms: An Example Based on Early's Recognizer," Proceedings of the Conference on Proving Assertions About Programs (January 1972), pp. 150-169.

Jones, L.H. "The Role of Instruction Sequencing in Structured Microprogramming," SIGMICRO Newsletter, vol. 4, no. 3 (October 1973), pp. 17-21.

Keefe, D.D. "Hierarchical Control Programs for Systems Evaluation," IBM Systems Journal, vol. 7, no. 2 (1968), pp. 123-133.

Knuth, D.E. "A Review of Structured Programming," National Technical Information Service (NTIS) PB 223 572/9WC.

Knuth, D.E. and Floyd, R.W. "Notes on Avoiding GOTO Statements," Computer Science Technical Report CS 148, Stanford University (January 1970).

Leavenworth, B.M. "Programming With(out) the GOTO," Proceedings of the ACM 25, vol. II, (1972), ACM, New York, pp. 782-786.

London, R.L. "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence 5, American Elsevier Publishing Company, New York (1945), pp. 569-580.

Luckman, D.C. and Park, D.M.R. "On Formalized Computer Programs," Journal of Computer and System Sciences (June 1970), pp. 220-249.

Magnusan, R.A. "A Structured Assembly Language Source Program Generator, Version 4," National Technical Information Service (NTIS) PN-225-094/2WC (1973).

Manna, Z. and Ashcroft, E.A. "The Translation of GOTO Programs to WHILE Programs," Stanford University, California, CS-188 (1970).

McCracken, D.D. "Revolution in Programming," Datamation, vol. 19 (December 1973), pp. 50-52.

McGonagle, J.D. "A Study of a Software Development Project, Final Report," Contract No. F04701-71-C-0373, CCIP-85, Air Force Systems Command, Los Angeles (1971).

McHenry, R.C. "Management Concepts for Top Down Structured Programming," IBM Corporation, Gaithersburg, Maryland, FSC 73-001 (February 1973).

Miller, E. "Bibliography of Practical Software Validation Techniques," General Research Corporation, Santa Barbara, California.

Miller, E.F. "Extensions to FORTRAN and Structured Programming - An Experiment," General Research Corporation, Santa Barbara, California, RM-1608 (March 1972).

Miller, E.F. and Lindamood, G.E. "Structured Programming Top Down Approach," Datamation, vol. 19 (December 1973), pp. 55-57.

Mills, H. D. "Mathematical Foundations for Structured Programming," IBM Corporation, Gaithersburg, Maryland, FSC 72-6012 (February 1972).

Mills, H. D. "On the Development of Large Reliable Programs," SIGPLAN Notices, vol. 8, no. 3 (August 1973), p. 2.

Mills, H. D. "Top Down Programming in Large Systems," Debugging Techniques in Large Systems, (Rustin, R., ed.), Prentice Hall, New York (1971), pp. 41-56.

Mills, H. D. "Syntax-Directed Documentation for PL 360," Communications of the ACM, vol. 13, no. 4 (1970), pp. 216-222.

Mills, H. D. "Structured Programming in Large Systems," unpublished memo (November 1970).

Nassi, I. and Shneiderman, B. "Flowchart Techniques for Structured Programming," SIGPLAN Notices, vol. 8, no. 3 (August 1973), pp. 12-26.

Naur, P. "An Experiment on Program Development," BIT, vol. 12 (1972), pp. 347-365.

Naur, P. "Programming by Action Cluster," BIT, vol. 9 (1969), pp. 250-258.

Ogden, J. L. "Improving Software Reliability," Datamation, vol. 19 (January 1973), pp. 49-52.

Pagan, F. G. "On the Teaching of Disciplined Programming," SIGPLAN Notices, vol. 8 (October 1973), pp. 44-48.

Parnas, D. L. and Darringer, J. "SODAS and a Methodology for Systems Design," Proceedings FJCC (1967), pp. 449-474.

Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, vol. 15 (December 1972), pp. 1053-1058.

Parnas, D. L. "More on Simulation Languages and Design Methodology for Computer Systems," Proceedings FJCC (1969), pp. 739-743.

Parnas, D. L. "Sequential Process: A Fuzzy Concept," unpublished memo available on request from the author.

Parnas, D. L. "A Language for Describing the Function of Synchronous Systems," Communications of the ACM, vol. 9, no. 2 (1966).

Parnas, D. L. "A Paradigm for Software Module Specification with Examples," Carnegie-Mellon University, Department of Computer Science (March 1971).

- Randell, B. "Toward a Methodology of Computing System Design," Software Engineering, NATO Science Affairs Division, Brussels 39 (October 1968), pp. 204-208.
- Schnupp, P. "Abstraction and Model Formation - Toois of the Software Developer," ONLINE (Germany), vol. 10, no. 78 (August 1973), pp. 526-532.
- Snowden, R. A. "System for the Preparation and Validation of Structured Programs," Program Test Methods (Hetzel, W. C., ed.), Prentice-Hall, Englewood Cliffs, New Jersey (1973), pp. 57-54.
- Stillman, R. B. "A Survey of Techniques for Increasing Software Reliability," Proceedings of 1973 Summer Computation Simulation Conference, II, LaJolla, California, U.S.A. Simulation Council (1973), pp. 1130-1133.
- Varney, R. C. and Gotterer, M. H. "The Structural Foundation for an Operating System," Computer Journal, vol. 16, no. 4 (November 1973).
- Wegner, E. "Tree Structured Programs," Communications of the ACM, vol. 16, (November 1973), pp. 704-705.
- Weinberg, G. M. "The Psychology of Computer Programming," Van Nostrand (1971).
- Weinberg, G. M. "The Psychology of Improved Programming Performance," Datamation, vol. 18 (November 1972), pp. 82-85.
- Wirth, N. "Program Development by Stepwise Refinement," Communications of the ACM, vol. 14 (April 1971), pp. 221-227.
- Wirth, N. Systematic Programming: An Introduction, Prentice-Hall, Englewood Cliffs, New Jersey (1973).
- Wulf, W. A. "A Case Against the GOTO," Proceedings of the ACM 25, vol. II (1972), ACM, New York, pp. 791-796.
- Zurcher, F. W. and Randell, B. "Multi-Level Modeling - A Methodology for Computer System Design," Proceedings IFIPS (1968).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM												
1. REPORT NUMBER AFAL-TR-73-40	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER												
4. TITLE (and Subtitle) Structured Software Study.	5. TYPE OF REPORT & PERIOD COVERED Final Rept. 1 May 72 - Feb 74.	6. PERFORMING ORG. REPORT NUMBER JDM 73-5												
7. AUTHOR(s) J. David McGonagle	8. CONTRACT OR GRANT NUMBER(s) F33615-72-C-1972													
9. PERFORMING ORGANIZATION NAME AND ADDRESS J. D. McGonagle and Company 246 Howarth Road Media, Pennsylvania 19063	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 6090													
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Avionics Laboratory (FY1175) AFAL/ AAT Wright Patterson AFB, Ohio 45433	12. REPORT DATE 31 October 1974													
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	13. NUMBER OF PAGES 96													
	15. SECURITY CLASS. (of this report) UNCLASSIFIED													
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A													
16. DISTRIBUTION STATEMENT (of this Report) Distribution limited to U. S. Government agencies only, Test and Evaluation, 19 July 1974. Other requests for this document must be referred to the Air Force Avionics Laboratory (AAM), Wright-Patterson AFB, Ohio 45433.														
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) F33615-72-C-1972														
18. SUPPLEMENTARY NOTES 6090 / 17, 91														
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) <table border="0"> <tr> <td>Structured Programming</td> <td>Program Decomposition</td> <td>PDP-11</td> </tr> <tr> <td>Program Structure</td> <td>Modular Programming</td> <td>CPU Simulation</td> </tr> <tr> <td>Program Design</td> <td>Top-Down Design</td> <td>Program Validity</td> </tr> <tr> <td>Program Composition</td> <td>Understanding Programming</td> <td></td> </tr> </table>			Structured Programming	Program Decomposition	PDP-11	Program Structure	Modular Programming	CPU Simulation	Program Design	Top-Down Design	Program Validity	Program Composition	Understanding Programming	
Structured Programming	Program Decomposition	PDP-11												
Program Structure	Modular Programming	CPU Simulation												
Program Design	Top-Down Design	Program Validity												
Program Composition	Understanding Programming													
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This study reports an evaluation of Structured Programming as an aid in the production of highly reliable computer programs. An approach to problem analysis and program composition which organizes the program text to clearly reflect the order of execution for the program. The resultant program text reflects the sub-division of the problem into smaller tasks which are clearly identifiable. The														

DD FORM 1473 1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

405358 ✓

over
P

7/B

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

rules for performance and ordering of these sub-tasks are reflected in the limited but sufficient set of controls used in the program construction. A set of Principles of Structured Programming are developed together with guides for determining an optimal upper and lower bounds on a program size. The applications of the Principles to a program are illustrated in the study report. A set of observations and conclusions drawn from the experience of developing a program in this way are presented. A bibliography of structured programming is reported.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

END 2-77