A TOP-DOWN STRUCTURED PROGRAMMING
TECHNIQUE FOR MINI-COMPUTERS

THESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of
MASTER OF SCIENCE

by

Robert Chin-yi Wu, B. S., M. S.

Denton, Texas
May, 1978

This paper reviews numerous theoretical results on control structures and demonstrates their practical examples. This study deals with the design of run-time support routines by using top-down structured programming technique. A number of examples are given as illustration of this method. In conclusion, structured programming has proved to be an important methodology for systematic program design and development.
<table>
<thead>
<tr>
<th>Letter(s)</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Character Variable</td>
<td>89</td>
</tr>
<tr>
<td>DCL</td>
<td>Declare statement</td>
<td>29</td>
</tr>
<tr>
<td>INIT</td>
<td>An initial value to be assigned to a variable</td>
<td>89</td>
</tr>
<tr>
<td>N</td>
<td>Numeric variable</td>
<td>89</td>
</tr>
<tr>
<td>OP1</td>
<td>first operand</td>
<td>25</td>
</tr>
<tr>
<td>OP2</td>
<td>second operand</td>
<td>25</td>
</tr>
<tr>
<td>OP3</td>
<td>third operand</td>
<td>25</td>
</tr>
<tr>
<td>VC</td>
<td>Character variable with variable-length</td>
<td>89</td>
</tr>
<tr>
<td>VN</td>
<td>Numeric variable with variable-length</td>
<td>89</td>
</tr>
<tr>
<td>WKA1</td>
<td>first intermediate work area</td>
<td>45</td>
</tr>
<tr>
<td>WKA2</td>
<td>second intermediate work area</td>
<td>45</td>
</tr>
<tr>
<td>WKA3</td>
<td>third intermediate work area</td>
<td>45</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE PAGE</td>
<td>i</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>GLOSSARY OF SYMBOLS</td>
<td>iii</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
</tbody>
</table>

Chapter

I. INTRODUCTION ........................................ 1

II. STRUCTURED PROGRAMMING .............................. 9

- What Is Structured Programming?
- Why Structured Programming
- Definition Of Structured Programming
- Related Work To Structured Programming

III. THE BASIC STRUCTURE OF A COMPILER .............. 20

- Description Of A Compiler-interpreter System
- Addressing
- Data Type
- Data Attribute
# TABLE OF CONTENTS (Continued)

IV. A TECHNIQUE OF TOP-DOWN STRUCTURED PROGRAMMING FOR ASSEMBLY LANGUAGES ........................................ 34

- Block Structure
- LOOP-EXIT-END and REPEAT-END
- EXIT and EXITLIST
- BEGIN-END
- IF-THEN-ELSE and CASE-ELSE
- SUBROUTINE-END
- Summary

V. TOP-DOWN STRUCTURED PROGRAMMING DESIGN FOR RUN-TIME SUPPORT ROUTINES ........................................ 43

- The Process Of Data Manipulation
- Discussion Of The Subtraction Process
- Discussion Of The Multiplication Process
- Discussion Of The Division Process

VI. CONCLUSION .................................................. 72

APPENDIX I .................................................... 77

APPENDIX II ................................................... 89

BIBLIOGRAPHY .................................................. 90
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Unstructured Flowchart</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>Structured Flowchart</td>
<td>5</td>
</tr>
<tr>
<td>3.</td>
<td>Unstructured Flowchart Of A Program</td>
<td>6</td>
</tr>
<tr>
<td>4.</td>
<td>Structured Flowchart Of A Program</td>
<td>7</td>
</tr>
<tr>
<td>5.</td>
<td>Linear Sequence Structure</td>
<td>11</td>
</tr>
<tr>
<td>6.</td>
<td>Iterative Structure</td>
<td>11</td>
</tr>
<tr>
<td>7.</td>
<td>Selective Structure - IF</td>
<td>12</td>
</tr>
<tr>
<td>8.</td>
<td>Selective Structure - CASE</td>
<td>12</td>
</tr>
<tr>
<td>9.</td>
<td>The Structure Of A Compiler System Mechanism</td>
<td>26</td>
</tr>
<tr>
<td>10.</td>
<td>Indirect Addressing</td>
<td>27</td>
</tr>
<tr>
<td>11.</td>
<td>List Structure For The Data Area</td>
<td>28</td>
</tr>
<tr>
<td>12.</td>
<td>Description Of The Data Attribute</td>
<td>32</td>
</tr>
<tr>
<td>13.</td>
<td>Structured Flowchart Of The Addition Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>14.</td>
<td>Structured Flowchart Of The Subtraction Algorithm</td>
<td>52</td>
</tr>
<tr>
<td>15.</td>
<td>Structured Flowchart Of The Addition/Subtraction Algorithm</td>
<td>53</td>
</tr>
<tr>
<td>16.</td>
<td>A Table For The Multiplication-table Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>17.</td>
<td>A Diagram For Multiplication-table Method</td>
<td>58</td>
</tr>
<tr>
<td>18.</td>
<td>A Diagram For Multiplication-table Method</td>
<td>59</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Two-address Format</td>
<td>25</td>
</tr>
<tr>
<td>2. Three-address Format</td>
<td>25</td>
</tr>
<tr>
<td>3. A Diagram For Repeated Subtraction In</td>
<td>71</td>
</tr>
<tr>
<td>The Division</td>
<td></td>
</tr>
<tr>
<td>4. Structured Programming Results From</td>
<td>75</td>
</tr>
<tr>
<td>The Interpreter</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

One of the questions that is often asked is what is the difference between unstructured and structured programming? A computing problem is often first expressed in a flowchart form (1), from which detailed coding is prepared at a later stage. Flowcharts are so extensively used that it has been suggested that they be used for entering programs directly into a computing system. A flowchart notation is now widely being used for describing algorithms (2). The first proof that structured programs were sufficiently powerful to represent any flowchartable program logic was due to Bohm and Jacopini (1). Therefore, we can distinguish the unstructured and structured programming by examining a flowchart. There are two examples shown in Figures 1, 2, 3 and 4.

Mills (6, 7, 8) has defined a proper program as one whose flowchart has precisely one single-entry and single-exit control structure, and for each node in the flowchart, there is a path from the entry line through the functional block (or processing block) to the exit line. Bohm and Jacopini (1) have shown that every proper program can be
effectively translated into an equivalent flowchart which uses only linear sequence, IF-THEN-ELSE and DO-WHILE flow-of-control forms. Structured programming is built upon these flow-of-control forms as below:

(1) linear sequences : concatenation of each block
(2) iteration : repetition such as LOOP, REPEAT or DO-WHILE
(3) selection : IF-THEN-ELSE, CASE-ELSE

These structures are illustrated and discussed in the next chapter. In this paper, the term "structured programming" will be used to refer to the use of the concept of Mills, Dijkstra, Bohm and Jacopini to build up a program from a small set of readily understood control structures. This is an approach to solve problems using top-down, hierarchical method, which is illustrated by Wirth (9) and McGowan & Kelly (4). The reason for choosing this technique is described in Chapter II. More details to apply this technique to the design of a program are discussed in Chapter IV.

Chapter III describes briefly the general concept of a compiler. It is illustrated in the method of addressing (5) and the concept of paging (3). The definition of data type (10) is precisely specified in Section III.3 & 4.

This paper will focus on the construction of compiler
run-time support routines by using top-down structured programming. Each of these will be a macro PROCEDURE-END in this design. There may be several SUBROUTINES in a PROCEDURE to eliminate duplication of codes. The design of compiler run-time support routines is described in Chapter V.

In a commercial application language system, it was found that all aspects of the structured approach: top-down development, modular design and structured walk-through contributed to a much more efficient overall project control.
Figure 1. Unstructured Flowchart
Figure 2. Structured Flowchart
Figure 3. Unstructured Flowchart Of A Program
Figure 4. Structured Flowchart of A Program
CHAPTER BIBLIOGRAPHY


CHAPTER II
STRUCTURED PROGRAMMING

This chapter surveys the major concepts associated with structured programming. What is structured programming? What has caused structured programming to be important? What are recent studies related to the top-down structured design concept? These are the questions directed to a programmer who wants to employ this methodology for systematic program design and development.

II.1 What is structured programming?

Dijkstra first published (4) a paper on the subject of structured programming in 1965. He later pointed out (5) that the GO TO construct should be abolished from all high-level programming languages. He also indicated the fact that the GO TO statement is unnecessary and can be replaced by conditional, WHILE and REPEAT clauses.

Bohm and Jacopini showed in 1966 that every program can be logically defined using only three flow-of-control structures (2). They are 1) linear sequences, 2) iteration, 3) selection. These structures are illustrated in Figures 5, 6, 7 and 8.
The linear sequences, shown in Figure 5, represents the concatenation of a functional block. The iterative structure, illustrated in Figure 6, causes the repetitive execution of the entire block. The loop exit arises from the exceptional condition at the top of the LOOP form and the termination test comes at the end of each execution of the block in the REPEAT form. More detail will be given in Chapter IV. A selective structure, given in Figure 7, causes the execution of a particular block of instructions depending on the selective mechanism. When the IF-THEN-ELSE statement is true, all instructions within the THEN block are executed, otherwise all instructions within the ELSE block are executed. The CASE statement (19), illustrated in Figure 8 is an extension of the IF-THEN-ELSE structure. There are more than two consecutive selective possibilities depending upon the selective mechanism. Other programming structures such as the WHILE-UNTIL (7) and DO-ENDO (3) constructs have been introduced in an attempt to increase the number of tools available to the programmer above the minimal set of the three previous mentioned.

Structured programs which are identified as compound functions were described in the articles of Mills (14, 16) and Ledgard (10) in 1975. They showed how the various functions comprise a program and how to utilize certain
Figure 5. Linear Sequence Structure

Figure 6. Iterative Structure
Figure 7. Selective Structure - IF

Figure 8. Selective Structure - CASE
axioms to prove a program to be correct.

Mills (13) was more interested in achieving a hierachical tree structure of single-entry and single-exit in 1971. He described "structured programming" not as the absence of the GO TO statement, but the presence of structure in 1973 (14). He emphasized a proof of correctness in program design in 1975 (15).

Top-down structured program design (11, 13) is an approach to use a hierarchical structure upon the program. It begins with a design of major functions and their interfaces, and breaking those functions into successively smaller sub-functions (20). This method can be applied recursively by splitting the sub-functions into more detailed actions (21).

A variety of other papers have been published in the literature (6, 12, 17). Connally's Master Thesis (3) has given more a detailed in his survey of structured programming.

II.2 Why structured programming?

(1) Structured programming produces a better documented program with greater readability (see Appendix I). A programmer can read a program segment from top to bottom without having to follow through undisciplined transfers of
control such as GO TO statement or any BRANCH instruction.

(2) Structured programs should be easier to debug and understand. IBM published the results of applying this methodology (6) to the design of the New York Times project in January, 1972. Its reports showed that programmer productivity can be greatly increased and that coding error rates can be greatly reduced (one detected error per 10,000 lines of coding). GTE announced in Computer World, 1975 that this approach to structured programming has enabled this company to reduce debug time from 50% of total programming time to practically zero. There can be fantastic results from this systematic program design and development approach.

(3) Structured programming offers a new concept of the constructing design. It helps the user develop distinct clearly defined program logic modules.

(4) Modification can be made to isolated blocks of code without affecting other major segments of the programs. The development of this methodology has been motivated by a desire to reduce the cost of developing and maintaining software.
This technique will be
1) simple in the concept and the presentation,
2) readily understood control flow of the programming process,
3) easily translated to computer code from structured flowchart.

II.3 Definition of structured programming

A summary of the definition of structured programming is given as below.

1) It is top-down programming that makes the programs easily understood and modified (6, 11, 13).
2) Its flowchart forms are satisfied with the basic Bohm and Jacopini mechanism (2) and Mills (13) one-entry & one-exit concept.
3) It restricts the use of an undisciplined GO TO statement or any BRANCH instruction to jump around in a program (5, 9).
4) It rearranges the program logic concept, program notation to maintain program correctness (8, 15).
5) It changes the control structures to code for iteration and selection (2, 6, 11).
6) It supports stepwise refinement process at "small" program level, which is no longer than one listing
II.4 Related work to structured programming

McGowan and Kelly (11) have developed the philosophy and technique of top-down structured programming in their textbook. In their book they apply this methodology to demonstrate the structured programming preprocessor for PL/1 language. A sequence of hierarchical semantic models is recommended for designing the language and its compiler by Basili (1). An experimental structured programming language, STAPLE, with nested block structure in the source language is written to indicate flow of control by Stewart (18).

Connally (3) developed the IBM "DO . . . ENDO" & "CASE . . . ENDCASE" macros to help a programmer to employ structured programming throughout his assembly language coding. Friedman and Shapiro (7) introduced a syntactic combination of the WHILE and the UNTIL macro in their paper. They illustrated examples to indicate that use of the WHILE-UNTIL can lead to more manageable structured programming.

Most of them didn't illustrated a real structured program which utilizes the macros and demonstrated the control structures of a designing system. Therefore, the
author chose to use the macros, which are described in Chapter IV to design mini-computer compiler run-time routines.

The author has found that all facts of the structured approach - top-down development modular design and structured walk-through contribute to a much more efficient overall project control.
CHAPTER BIBLIOGRAPHY


CHAPTER III
THE BASIC STRUCTURE OF A COMPILER

This chapter describes briefly the general concept of a compiler, which is shown in Figure 9. For the purpose of examples, attention is focused on the run-time support routines for arithmetic operation in a commercial application language.

In order for the run-time support routines to perform their designated actions properly, several areas need to be discussed.

First, we should know where to get data in each of the operands before doing the arithmetic operations. Therefore, we have to understand the addressing mechanism and the internal text in a core map.

Second, we need to know what the data type of each operand is. The question is where to get it. The data type accessing mechanism is outlined in Figure 9 and will be discussed later. Some of the data attributes that are extracted are numeric/character, left/right justify, zero or blank fill and fixed/variable length (see Section III.4). Then, they are stored in intermediate data-attribute work area for later use by the run-time support routines.
Third, we cannot handle any arithmetic before the data are moved to intermediate work area. The data items are scanned for validity according to their data attributes. We need to build up a program subroutine to transfer all of the data from the object program's data area into an intermediate work area for arithmetic operations.

III.1 Description of a compiler-interpreter system

A compiler is a program which takes as data the program being compiled (called the source program) and produces as its result, an equivalent program in a binary machine language (called the object program), as shown in Figure 4. These processes are listed below.

1) The compiler routines initially lexically scan the source program by transforming it into more easily recognized symbols. Next, a syntactic analysis is performed to recognize the basic valid constructs and to create the symbol table.

2) A final compiler phase consists of the code generation. This involves generating a binary "pseudo" machine language for those statements which are syntactically and semantically correct (1).

3) The generated code is then placed in a mass storage memory for later use.
4) When the execution of the program is requested, the program is made available. This involves the interpreter which emulates a virtual machine which performs the actions specified in the pseudo machine language which was generated by the compiler.

5) The interpreter interacts with the I/O control operations such as DISPLAY, ACCEPT, PRINT, WRITE or READ and so forth. Most of arithmetic operations will be done here. The final result will be transferred into the target data area.

6) As a pseudo instruction is executed, additional instructions are made available to the interpreter as required.

III.2 Addressing

The pseudo machine language produced by the compiler is placed in main storage when required. A fetch-instruction subroutine accesses an instruction from main storage and then positions a pointer at the beginning of the next sequential statement. A decode routine scans the operation field of the intermediate text statement. The formats of the various instructions are listed below.

(1) The two-address format
It is composed of an operation code and one or two operands. The basic structure of the text is

<table>
<thead>
<tr>
<th># of ops</th>
<th>1st op</th>
<th>2nd op</th>
</tr>
</thead>
<tbody>
<tr>
<td>op code</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2) The three-address format

It is composed of an operation code, and three operands. The basic structure of the text is

<table>
<thead>
<tr>
<th># of ops</th>
<th>1st op</th>
<th>2nd op</th>
<th>3rd op</th>
</tr>
</thead>
<tbody>
<tr>
<td>op code</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each of the operands is specified either by an address or by a pointer to the address. The operations performed using these formats are shown in Tables 1 and 2.

(3) The five-address format is

<table>
<thead>
<tr>
<th># of ops</th>
<th>1st op</th>
<th>2nd op</th>
<th>3rd op</th>
<th>4th op</th>
<th>5th op</th>
</tr>
</thead>
<tbody>
<tr>
<td>op code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: MVCHAR OPl,TWO,'MOVE A STRING',ONE,SEVEN

The design of data area is an important part of compiler construction since it provides the bridge between syntax and
code generation. It will be shown in Figures 10 and 11.

Figure 10 illustrates the method of addressing (3) used to access data from compiled program. An instruction consists of header information followed by pointers to the appropriate operands. The header information consists of an indicator which describes the length of the operand list and an operation-code which specifies the operation to be performed. Each operand of an instruction consists of an attribute-address pair which points to the attribute of the data item being referenced and the contents of the data item respectively. The attribute of a data item completely describes the data item, i.e. its data type (numeric or character), its composition (array or structure) and its length (physical length or logical length).

In Figure 11 the design of the data area is extended with the concept of paging (2). This allows for a more optimal usage of main memory since a data area for a subroutine need not be in main memory if it is not executing.

Incorporated in this paging mechanism are several descriptors which did not exist earlier. A header record is used to describe this page and to provide a link to the next page. lst_text is the starting address of the first attribute-value pair in this page (other items may appear
<table>
<thead>
<tr>
<th>op code</th>
<th>operands</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/SUB</td>
<td>OP1,OP2</td>
<td>OP1 ← OP1 + or - OP2</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>OP1,OP2</td>
<td>OP1 ← OP1 * or / OP2</td>
</tr>
</tbody>
</table>

Table 1. Two-address Format

<table>
<thead>
<tr>
<th>op code</th>
<th>operands</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/SUB</td>
<td>OP1,OP2,OP3</td>
<td>OP1 ← OP2 + or - OP3</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>OP1,OP2,OP3</td>
<td>OP1 ← OP2 * or / OP3</td>
</tr>
</tbody>
</table>

Table 2. Three-address Format
Figure 9. The Structure Of A Compiler
System Mechanism
Figure 10. Indirect Addressing
Figure 11. List Structure For The Data Area
at the beginning of a page). Last_text is the starting address of the last attribute-value pair in this page. Next_page is used to point to the next page.

III.3 Data Type

One of most important current software issues the reliability for a commercial application language. The interface between the type definition and the users of the type must be precisely specified. A concept of data type allows the programmer to identify the data abstraction being employed and defines a finite set of operation which are valid for that type definition (4).

Each data item can be classified into a category called type. This category describes the overall properties of the data item. The type of data allowed can be classified as arithmetic type or character type. We recommend that all variables be declared in the heading of a program. Such a variable declaration is denoted by

\[ \text{VARIABLE DCL TYPE,ATTRIBUTE} \]

These two classes of data type are described as below:

(A) **Arithmetic Data**

An arithmetic data item is one that has a numeric value. It has the characteristics (or data attribute later) of scale-factor and precision as specified in the
declaration. There are two types for arithmetic data:

(A.1) Numeric variables

1) Type integer
   This type represents the set of whole numeric strings, sign and blanks.

2) Type real
   This type represents the set of whole numeric strings, blanks, decimal-point and sign.

(A.2) Constants
   It covers the actual digits, sign and decimal-point if any.

Although in a mathematical sense the integers are a subset of the real numbers, in our computer system convention all numbers denoted with decimal-point (or nonzero scale-factor) are assumed to be type real.

(B) Character Data

This type of data item denotes a finite, ordered set of characters including whole numeric strings and special characters used by the computer system. It covers character variables and literals:

(B.1) Character variables
   This type represents the set of valid characters.

(B.2) Literals
   These are legal characters in the data set (or
null string) enclosed in quotation marks. If the string is desired to represent a quotation mark within the literal, it must appear as two immediately adjacent quotation marks.

III.4 Data Attribute

The properties or characteristics of numeric or character strings are given by the name of the data attributes as indicated in Figure 12. Data attributes are classified into two general classes, character and numeric, which may be simple, array or structure.

Most attributes may be determined during the parsing of the declaration statement (see Appendix II).
Figure 12. Description of the Data Attribute
CHAPTER BIBLIOGRAPHY


CHAPTER IV
A TECHNIQUE OF TOP DOWN STRUCTURED PROGRAMMING FOR ASSEMBLY LANGUAGES

Every programmer knows that an assembly language program can be more difficult to read, code, and debug than the same program in a suitable high-level language. If these structured programming concepts are applied to assembly languages then, the author believes, a program could be more easily implemented in assembly language.

Assembly language should provide block constructs that are available in high-level languages to facilitate structuring with greater understandability. It encourages or directs programmers to write their programs in a more structured manner. For instance, since we know that IF-THEN-ELSE and BEGIN-END will occur very frequently, we should make this construct as convenient and reliable as possible.

Structured programming is by no means restricted to high-level languages such COBOL, FORTRAN, and PL/1 and so forth. A well-organized programmer can write a well-structured program in assembly language, by using block construct of macros.
In this chapter, the author would like to recommend macros to develop the top-down structured program. They are shown as below:

(1) PROCEDURE-END,
(2) REPEAT-END,
(3) LOOP-END,
(4) SUBROUTINE-END,
(5) BEGIN-END,
(6) IF-THEN-ELSE,
(7) CASE-ELSE,
(8) CALL,
(9) EXECUTE,
(10) EXIT, and
(11) EXITLIST.

The use of these macros is described in the Harris Assembly Language User's Manual (3).
IV.1 Block structures

When using the assembly language, a program must consist of a single block with other blocks optionally nested within it. Every statement must appear within a block. There are six kinds of blocks existing in macros:

1. begin block,
2. loop block,
3. repeat block,
4. case block,
5. subroutine block, and
6. procedure block.

Each of the six blocks is delimited by a respective block header and an END statement. A procedure block is the major body of code to be assembled as single independent unit in a program. Every begin block, loop block, repeat block, subroutine block, and case block must be contained within some procedure block. An example of each block is shown in Appendix I.

IV.2 LOOP-EXIT-END and REPEAT-END

The REPEAT-END and LOOP-END figures, shown in Figure 3b are similar to DO-WHILE figure in that it allows a block of codes to be executed repeatedly. They are closed by an END pseudo-operation with a matching name of the label
name. With REPEAT the contained block is always executed at least one time. The termination test comes at the end of each execution of the block.

The LOOP-END figure is a little different than the REPEAT-END figure in Figure 6. It allows a block of code to be repeatedly executed until one of the termination indicators, either EXIT or EXITLIST is found.

IV.3 EXIT and EXITLIST-END

The EXIT figure causes control to advance to one of a set of predefined exits or to leave the entire block structure, which is one of the six possible blocks. A particular predefined exit can be specified in an EXIT statement and presumably will perform some termination action before the entire block is left. For an example of an EXITLIST construct sees Appendix I.

Our proposed assembly language would not have the "BRANCH" statement. The cumbersomeness of not having a BRANCH statement can be eliminated by introducing a restricted form of BRANCH in the form of an EXIT. The use of exits will require the introduction of labels to identify where control is to proceed to. In Appendix I an example of the case block is a good example for EXIT and EXITLIST figures.

An EXIT statement is used only in connection with any
block structure. The EXIT must be physically within the block that is being terminated. This means that control may only advance forward to specified locations.

The EXITLIST pseudo-operation indicates the beginning of a list of statements, one of which may be executed before exiting the block. The list is closed by an END pseudo-operation statement for that block.

IV.4 BEGIN-END

The begin block is delimited at the top by a BEGIN statement and a corresponding END statement at the end. Begin blocks are executed in the normal flow of a program, either sequentially or as a result of an EXITLIST or in an IF-THEN-ELSE or CASE-ELSE statement transfer. Each begin block must be nested within one of the six kinds of blocks. It is also demonstrated in an example of case block in Appendix I.

IV.5 IF-THEN-ELSE and CASE-ELSE

The IF-THEN-ELSE figure, shown in Figure 3c provides the common condition branch and is usually used with a simple comparison expression. It is shown as below:

```
IF condition
  THEN action-if-true;
  ELSE action-if-false.
```
where the logical conditions will be

(i) EQUAL/NOT_EQUAL,
(ii) ZERO/NOT_ZERO,
(iii) LOW/NOT_LOW,
(iv) HIGH/NOT_HIGH,
(v) PLUS/NOT_PLUS,
(vi) MINUS/NOT_MINUS, or
(vii) TRUE/FALSE.

It is held generally true that when IF statements are nested to several levels they are difficult to understand. Gildersleeve (2) mentioned that it is possible to employ flags and a decision table to produce a structured flowchart. Structured programming may survive. Bloom (1) also showed the "functional" programming from the decision table. By compounding conditions and using logical operators OR & AND, the nested IF can be eliminated or reduced to a more understandable level. An example of IF-THEN-ELSE in Appendix I.6 demonstrates this idea.

The case block figure, shown in Figure 8, is an extension of the IF-THEN-ELSE statement. In the CASE statement, in an assembly language, it is best to use a value in a particular index register to determine which statement (or block) within the CASE is to be executed. The integer in the index register would have a one-to-one
correspondence to the statements within the CASE structure. The index register is checked for a range before execution and if it is out of range, control will proceed to the ELSE statement.

IV.6 SUBROUTINE-END

A subroutine block may be invoked by an EXECUTE macro statement that names an entry point of the subroutine. A procedure block may be invoked by a CALL macro statement that names an entry point of the procedure, which is demonstrated in an example of ROUND subroutine in Appendix I.4. These two constructs allow for communication with external and internal programs.

IV.7 Summary

The author would like to illustrate the method for top-down structured programming as follows:

(1) One needs to draw a hierarchy chart for a single module.

(2) One needs to draw HIPO (Hierarchy, Plus Input, Process, Output) diagram to help in the understanding of the developing specification.

(3) One needs to draw another structured chart to get the module interface on the hierarchy chart.

(4) A programmer must make a structured flowchart for
each module. The program can then be coded by using block structures of macros from the structured flowchart. The program should be coded from bottom, but designed from top to bottom.
CHAPTER BIBLIOGRAPHY


Arithmetic operations involve the process of data manipulation. More detail of this process will be described in Section V.1. The basic arithmetic operations to be described are addition, subtraction, multiplication, and division.

In this chapter, both integers and real numbers are called fixed-point numbers. Fixed-point fractional arithmetic is the most widely used in arithmetic operations, although the radix point of a number does not physically exist inside the arithmetic intermediate work area after the data item of each operand is transferred over there. But the alignment of this radix point is set in this intermediate work area when the data item is moved there. Only fixed-point arithmetic is considered, since floating point arithmetic is not as useful for commercial applications.

In this chapter, algorithms for addition, subtraction, multiplication, and division will be developed utilizing the hexadecimal number representation, using IBM EBCDIC digits. Algorithms were designed to simulate th missing
hardware instructions such as MULTIPLY & DIVIDE and other hardware support. There is no need to be concerned with 1's or 2's complement arithmetic since all operations are directly under EBCDIC representation. They are described in Section V.1 and V.2.

The position of the radix point is very important in arithmetic operations. The radix point is especially important when processing the multiplication and the division of real numbers. They will be discussed in Section V.3 and V.4.
V.1 The process of data manipulation

Addition and subtraction of IBM EBCDIC numbers are very similar to that of hexadecimal binary numbers, since each EBCDIC digit is presented in binary form.

Discussion of the addition process

The EBCDIC numbers representing the augend and the addend are summed by adding each EBCDIC byte of the addend to the appropriate EBCDIC byte of the augend, starting with the lower (or the right-most) byte. The rules for adding two EBCDIC numbers are summarized in the following example. This is the algorithm for data conversion.

Examples of addition:

| BYTE of an augend from WKAl* | F7 | F7 |
| BYTE of an addend from WKA2* | F2 | F6 |
|                         |    |    |
| Subtract                | F0 | F0 |
| Adjustment              | F9 | FD |
| Subtract if greater than F9 |   | 0A |
| BYTE of result stored into WKA3* | F9 | F3 |
| Carry                   | 0  | 1  |

* See glossary of symbol.
Note: Case (1) is addition without carry;
Case (2) is addition with carry.

To do the arithmetic operation easily, let us strip out the decimal-point and sign from the intermediate work area. This means that all numbers we deal with are non-negative in the intermediate work area. The additional work of computing the sign of the result is quite straightforward under ordinary arithmetic rule of algebra.

ALGORITHM A (Addition of non-negative real numbers)

Given non-negative real numbers \( A_n \) and \( B_n \) with their signs \( A_s \) and \( B_s \) respectively, this algorithm forms their sum, \( R_n \) with its sign \( R_s \).

\( A1: \) If \( A_s \) is equal to \( B_s \), then \( R_n \) is the sum of \( A_n \) and \( B_n \);

\( R_s \) is the same as \( A_s \); and

the condition code is set depending on \( R_s \).

and the algorithm is terminated.

\( A2: \) If \( A_n \) is equal to \( B_n \), then \( R_n \) is zero;

\( R_s \) is plus-sign;

the condition code is "=".

and the algorithm is terminated.
A3: If $A_n > B_n$, then $R_n$ is equal to $(A_n - B_n)$;

$R_s$ is the same as $A_s$;
the condition code is set depending on $R_s$.
and the algorithm is terminated.

A4: $R_n$ is equal to $(B_n - A_n)$;

$R_s$ is the same as $B_s$;
the condition code is set depending on $R_s$;
and the algorithm is terminated.

Note: If the sign of $R_s$ is plus, then the condition code will be set to ";>"; otherwise it will be set to ";<". But if $R_n$ is zero, the condition code will be set to ";=".

Examples:

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>-3</td>
<td>-9</td>
<td>-8</td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-8</td>
<td>9</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>-11</td>
<td>0</td>
<td>-3</td>
<td>4</td>
</tr>
</tbody>
</table>

The structured flowchart of Algorithm A is shown in Figure 13.
Figure 13. Structured Flowchart of the Addition Algorithm
V.2 Discussion of the subtraction process

Subtraction introduces a new concept: borrowing. It can be executed directly under the EBCDIC byte of the minuend by subtracting from the EBCDIC byte of the subtrahend. The borrow--byte is always taken from next higher byte of the subtrahend.

I apply a method for the subtraction: the subtrahend must be smaller than the minuend. The sign of the minuend will be stored into a global data area before this operation.

Examples of subtraction:

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE of minuend in WKA1</td>
<td>F7</td>
<td>F2</td>
</tr>
<tr>
<td>BYTE of subtrahend in WKA2</td>
<td>F2</td>
<td>F7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F2</td>
</tr>
<tr>
<td>Adjustment</td>
<td>F0</td>
<td></td>
</tr>
<tr>
<td>BYTE of result stored into WKA3</td>
<td>F5</td>
<td>F5</td>
</tr>
</tbody>
</table>

Borrow 0
Sign +

Note: Case (1) is subtraction without borrowing;
      Case (2) is subtraction with borrowing.
ALGORITHM S (Subtraction of non-negative real number)

Given non-negative real numbers $A_n$ and $B_n$ with their $A_s$ and $B_s$ respectively, this algorithm forms their non-negative difference, $R_n$ with its sign $R_s$.

S1: If $A_s$ is not equal to $B_s$, then $R_n$ is the sum of $A_n$ and $B_n$;

$R_s$ is the same as $A_s$;

the condition code is set depending on $R_s$.

and the algorithm is terminated.

S2: If $A_n$ is equal to $B_n$, then $R_n$ is zero;

$R_s$ is plus-sign;

the condition code is set to "=".

and the algorithm is terminated.

S3: If $A_n > B_n$, then $R_n$ is the difference of $(A_n, B_n)$;

$R_s$ is the same as $A_s$;

the condition code is set depending on $R_s$.

and the algorithm is terminated.

S4: $R_n$ is the difference of $(B_n, A_n)$;

$R_s$ is the reverse sign of $B_s$, i.e. $-B_s$;

the condition code is set according to $R_s$;

and the algorithm is terminated.
Examples:  (1)  (2)  (3)  (4)  (5)
-8   -7   -5   -9   3
-8   -5   -7   9   7

The structured flowchart of Algorithm S is shown in Figure 14.

In an attempt to minimize computer memory, coding instructions and running time requirements, the algorithm of addition and subtraction are very similar each other. They are combined in a flowchart form as is shown in Figure 15. In this figure, OPl_ADD_OP2 is a subroutine to handle the sum of each byte of the augend and each byte of the ad-end. OPl.SUB.OP2 is a subroutine to get the difference between each byte of a larger operand and each byte of smaller one. This employs the process of data manipulation, which is discussed in Section V.1.
Figure 14. Structured Flowchart of the Subtraction Algorithm
Figure 15. Structured Flowchart of the
Addition/subtraction Algorithm
V.3 Discussion of the multiplication process

Multiplication is the process of adding a multiplicand to itself, m times, where m is the number representing the multiplier. The result will be the product of the multiplicand and the multiplier.

This can be accomplished by implementing these algorithms of multiplication below using

1) serial addition; or
2) multiplication table.

Example for serial addition method:

If given 456790 \times 74, then a 10\times 16 (if 16-byte length required) intermediate work area must be built, as shown below if the serial addition method is being used.

<table>
<thead>
<tr>
<th>Index</th>
<th>Work Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>0000 0000 0045 6790</td>
</tr>
<tr>
<td>2</td>
<td>0000 0000 0091 3580</td>
</tr>
<tr>
<td>3</td>
<td>0000 0000 0137 0370</td>
</tr>
<tr>
<td>4</td>
<td>0000 0000 0182 7160</td>
</tr>
<tr>
<td>5</td>
<td>0000 0000 0228 3950</td>
</tr>
<tr>
<td>6</td>
<td>0000 0000 0274 0740</td>
</tr>
<tr>
<td>7</td>
<td>0000 0000 0319 7530</td>
</tr>
<tr>
<td>8</td>
<td>0000 0000 0365 4320</td>
</tr>
<tr>
<td>9</td>
<td>0000 0000 0411 1110</td>
</tr>
</tbody>
</table>
The unsigned magnitude of two numeric operands is determined first by suppressing leading zeros (EBCDIC zero), then by comparing the strings one significant EBCDIC digit at one time. The larger data item of the operand is considered as the multiplicand and the smaller one as the multiplier.

The multiplicand is then placed in the intermediate work area so that the work area contains multiples of its ranging from one to nine. The multiplier is scanned from right to left one digit at time. The value of each digit of the multiplier (1 - 9) determines which value of the multiplicand is added into a summation area (WKA3), which it is illustrated below.

```
0000 0000 0182 7160 in WKA1
0000 0000 3197 5300 in WKA2
```
the result ▶ 0000 0000 3380 2460 in WKA3

With each subsequent digit of the multiplier, the pointer to working sum is moved left one digit when the multiplier has been scanned. The entire summation value in the third intermediate work area is moved to the target data area under alignment.
The algorithm for the multiplication-table method is illustrated below. First, it requires to have a multiplication table in Figure 16. Secondly, the authors applies the following formulas to evaluate the result of each digit from this 10X10 multiplication table. The formulas for the multiplication-table method is given by

1) CARRY = LEFT\_BYTE\_NUM + WKA\_CARRY;
2) WKA\_BYTE = RIGHT\_BYTE\_NUM + CARRY + WKA\_BYTE;
   \[ \begin{align*}
   &0 & \text{if } WKA\_BYTE < 10; \\
   &1 & \text{if } 10 \leq WKA\_BYTE < 20; \\
   &2 & \text{if } WKA\_BYTE \geq 20.
   \end{align*} \]

Example (1)

If given 1.09 X 4.82, the result after the multiplication is 5.2538. It is illustrated in Figure 17.

Example (2)

If given 78.9 X 7.9, the result after the multiplication is 623.31. It is illustrated in Figure 18.
**MULTIPLICAND**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10,X'00',X'00',X'00',X'00',X'00',X'00',X'00',X'00',X'00',X'00'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10,X'00',X'01',X'02',X'03',X'04',X'05',X'06',X'07',X'08',X'09'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10,X'00',X'02',X'04',X'06',X'08',X'10',X'12',X'14',X'16',X'18'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10,X'00',X'03',X'06',X'09',X'12',X'15',X'18',X'21',X'24',X'27'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10,X'00',X'04',X'08',X'12',X'16',X'20',X'24',X'28',X'32',X'36'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10,X'00',X'05',X'10',X'15',X'20',X'25',X'30',X'35',X'40',X'45'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10,X'00',X'06',X'12',X'18',X'24',X'30',X'36',X'42',X'48',X'54'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10,X'00',X'07',X'14',X'21',X'28',X'35',X'42',X'49',X'56',X'63'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10,X'00',X'08',X'16',X'24',X'32',X'40',X'48',X'56',X'64',X'72'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10,X'00',X'09',X'18',X'27',X'36',X'45',X'54',X'63',X'72',X'81'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 16. A Table For The Multiplication-Table Algorithm**
<table>
<thead>
<tr>
<th>Cycle</th>
<th>First Cycle</th>
<th>Second Cycle</th>
<th>Third Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9x2 0x2 1x2</td>
<td>9x8 0x8 1x8</td>
<td>9x4 0x4 1x4</td>
</tr>
<tr>
<td>Multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT_BYTE_NUM</td>
<td>0 1 0 0</td>
<td>0 7 0 0</td>
<td>0 3 0 0</td>
</tr>
<tr>
<td>RIGHT_BYTE_NUM</td>
<td>- 8 0 2</td>
<td>- 2 0 8</td>
<td>- 6 0 4</td>
</tr>
<tr>
<td>WKA_CARRY</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>CARRY</td>
<td>0 1 0 0</td>
<td>0 7 0 0</td>
<td>0 4 1 0</td>
</tr>
<tr>
<td>WKA_BYTE</td>
<td>F0 F8 F1 F2</td>
<td>F0 F3 F9 F8</td>
<td>F0 F5 F2 F5</td>
</tr>
</tbody>
</table>

right-most byte of WKA3

Figure 17. A diagram For Multiplication-table Method
<table>
<thead>
<tr>
<th>Cycle</th>
<th>First Cycle</th>
<th></th>
<th>Second Cycle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>9x9 8x9 7x9</td>
<td>9x7 8x9 7x7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT_BYTE_NUM</td>
<td>0 8 7 6</td>
<td>0 6 5 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RIGHT_BYTE_NUM</td>
<td>-1 2 3</td>
<td>-3 6 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WKA_CARRY</td>
<td>0 0 1 1</td>
<td>0 0 1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CARRY</td>
<td>0 8 8 7</td>
<td>0 6 6 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WKA_BYTE</td>
<td>F0 F1 F0 F1</td>
<td>F0 F3 F3 F2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 18.** A Diagram For The Multiplication-table Method
The method for the multiplication-table:

1) The multiplicand and multiplier are scanned from right to left one digit at time.

2) The product of each multiplicand-digit and multiplier-digit will be separated into two parts, of which one is LEFT_BYTE_NUM, and another is RIGHT_BYTE_NUM.

3) The result of each digit will be stored into a third intermediate work area (WKA3) for future use.

4) After each subsequent digit of the multiplicand is processed, the pointer to third work area is moved left one digit.

5) After a digit of the multiplier has been scanned, the entire summation value is moved to target data area according to declared data attribute.

The comparison between serial addition and multiplication-table method:

1) If a computer system allows 30-digit output limit, then it needs at least a 360-byte (30x9 + 3x30) intermediate work area. But it requires a 190-byte (10x10 + 3x30) work area for a multiplication-table method.

2) When a computer system builds the 30x9 work-area-
table by the serial addition method, it wastes a lot of execution time because of the initialization required.

3) Most mini-computer systems apply "serial addition" method to do this job because it is simple and easy to understand.

Summary of these two methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution time</th>
<th>Memory storage</th>
<th>Theory</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial addition</td>
<td>long</td>
<td>waste</td>
<td>simple</td>
<td>difficult</td>
</tr>
<tr>
<td>Multiplication table</td>
<td>short</td>
<td>10X10 bytes</td>
<td>Complicated</td>
<td>easy</td>
</tr>
</tbody>
</table>
V.4 Discussion of the division process

Division is the inverse of multiplication, so it is performed under subtraction. Most of the division techniques are a trial and error process, since after each subtraction the remainder must be tested for its magnitude. This method employs the technique of repeated subtraction and applies to the EBCDIC hexadecimal representation as described in Section V.2. It is the so-called the comparison method.

In this paper the author strips out the signs and the decimal-point of dividend and divisor before they are moved into the intermediate work area. The signs will be stored into working storage for later use, so the division is done using non-negative numbers. The data items of the dividend and the divisor are moved to first work area (WKAL) and second work area (WKA2) respectively. They are aligned under the same length of the fraction-portion of non-negative numbers, i.e. maximum scale-factor. This is illustrated in Step (1) of examples (1) and (2).

ALGORITHM D (Division of non-negative real number)

Given non-negative real numbers \( A_i \) and \( B_j \) with their signs \( A_s \) and \( B_s \) respectively, this algorithm forms the quotient \( Q_k \) and the remainder \( R \) with their signs \( Q_s \) and
The sign of the quotient, \( Q_s \) is determined by the rules of algebra:

1) \( Q_s \) is "plus", if \( A_s \) is like to \( B_s \);
2) \( Q_s \) is "minus", if \( A_s \) is unlike to \( B_s \).

The sign of the remainder, \( R_s \) is the same as that of the dividend except that zero quotient or zero remainder is set to positive.

D1: If \( B \) is zero, then set up an error message and terminate the program.

D2: If \( A \) is zero, then

1) the quotient \( Q \) is set to zero,
2) the condition code is set to "=",
3) the remainder \( R \) is zero, and
4) the signs of the quotient and the remainder are set to "plus".

D3: Initialize

1) \( i \leftarrow j + k + i + 1 \),
2) \( Q_k \leftarrow 0 \),
3) \( k \leftarrow 1 \).

D4: If \( i < j \), then \( Q_k \) is zero and terminate the program.

D5: If \( \sum A_j < \sum B_j \), then

1) \( k \leftarrow k + 1 \),
2) \( j \leftarrow j + 1 \),
3) \( Q_k \leftarrow 0 \),
and it forces to pad the leading zero(s) in front of the first existing B;

GO TO D4;

D6 : 1) $\Sigma R_j \leftarrow \Sigma A_j - \Sigma B_j$

2) $Q_k \leftarrow Q_k + 1,$

3) $\Sigma A_j \leftarrow \Sigma R_k$, and

GO TO D5.

Note: $\Sigma A_j$ represents $A_1 A_2 A_3 A_4$ if given $j = 4.$
Example (1):

If given 7415/4, and declared target'scale-factor=0, the division process will be:

Step (1) 0000 0000 0000 7415  
           BYTE of dividend in WKAl
0000 0000 0000 0004  
           BYTE of divisor in WKA2
0000 0000 0000 0000  
           BYTE of remainder in WKA3
0000 0000 0000 0000  
           BYTE of result stored into WKA4

1st comparison

Step (2) 0000 0000 0000 3415  
           BYTE of dividend in WKAl
0000 0000 0000 0004  
           BYTE of divisor in WKA2
0000 0000 0000 3415  
           BYTE of remainder in WKA3
1000 0000 0000 0000  
           BYTE of result stored into WKA4

2nd comparison

Step (3) 0000 0000 0000 0215  
           BYTE of dividend in WKAl
0000 0000 0000 0004  
           BYTE of divisor in WKA2
0000 0000 0000 0215  
           BYTE of remainder in WKA3
1800 0000 0000 0000  
           BYTE of result stored into WKA4

3rd comparison

Step (4) 0000 0000 0000 0015  
           BYTE of dividend in WKAl
0000 0000 0000 0004  
           BYTE of divisor in WKA2
0000 0000 0000 0015  
           BYTE of remainder in WKA3
1850 0000 0000 0000  
           BYTE of result stored
4th comparison

Step (5) 0000 0000 0000 0003  BYTE of dividend in WKA1
0000 0000 0000 0004  BYTE of divisor in WKA2
0000 0000 0000 0003  BYTE of remainder in WKA3
1853 0000 0000 0000  BYTE of result stored into WKA4

5th comparison

Step (6) 0000 0000 0000 0003  BYTE of dividend in WKA1
0000 0000 0000 0004  BYTE of divisor in WKA2
0000 0000 0000 1853  BYTE of remainder in WKA3
0000 0000 0000 0000  BYTE for rounding

The final result are (1) the quotient is 1853,

(2) the remainder is +3.
Discussion for second example:

(1) The data items of the dividend and the divisor are moved to first and second work areas respectively under \textit{radix point alignment}.

It requires zeros filled in first work area (WKAl) because the scale-factor of the declared target is 2. The author gives one more for rounding and truncation in this division. There is no rounding and truncation occurring in work area for integer target as can see in Example (1). The third \texttt{and} fourth work areas (WKA3 & WKA4) are initialized whole EBCDIC zero numbers in these fields.

(2) The comparison between the dividend and the divisor is performed the same number of digit-field. A technique of repeated subtraction is performed depending upon this comparison. Each digit of the quotient will be stored into a fourth work area (WKA4) from left to right fashion.

(3) We know that the digits of the minuend are in first work area (WKAl) and the digits of the subtrahend in second work area (WKA2) under subtraction (see Section V.2). The difference between the minuend and the subtrahend will be in the third work area (WKA3). Therefore, we must move entire data in third work area
to first work area after each complete subtraction, and fill all EBCDIC ZEROS in third work area after moving.

(4) After dividing completely, the data item in fourth work area (WKA4) should move to third work area under right alignment. Then, the rounding and truncation will be performed in Step (8).
Example (2) :

If given 74.15/0.4 and declared target's scale 2, the division process will be:

Step (1) 0000 0000 0741 5000   BYTE of dividend in WKAl
          0000 0000 0000 0040   BYTE of divisor in WKA2
          0000 0000 0000 0000   BYTE of WKA3
          0000 0000 0000 0000   BYTE of WLA4

                   1st comparison

Step (2) 0000 0000 0341 5000   BYTE of dividend in WKAl
          0000 0000 0000 0040   BYTE of divisor in WKA2
          0000 0000 0341 5000   BYTE of remainder WKA3
          1000 0000 0000 0000   BYTE of result stored into WKA4

                   2nd comparison

Step (3) 0000 0000 0021 5000   BYTE of dividend in WKAl
          0000 0000 0000 0040   BYTE of divisor in WKA2
          0000 0000 0021 5000   BYTE of remainder WKA3
          1800 0000 0000 0000   BYTE of result stored into WKA4

                   3rd comparison

Step (4) 0000 0000 0001 5000   BYTE of dividend in WKAl
          0000 0000 0000 0040   BYTE of divisor in WKA2
          0000 0000 0001 0040   BYTE of remainder in WKA3
          1850 0000 0000 0000   BYTE of result stored
into WKA4

4th comparison

Step (5) 0000 0000 0000 3000 BYTE of dividend in WKA1
       0000 0000 0000 0040 BYTE of divisor in WKA2
       0000 0000 0000 3000 BYTE of remainder in WKA3
       1853 0000 0000 0000; BYTE of result stored
       into WKA4

5th comparison

Step (6) 0000 0000 0000 0200 BYTE of dividend in WKA1
       0000 0000 0000 0040 BYTE of divisor in WKA2
       0000 0000 0000 0200 BYTE of remainder in WKA3
       1853 7000 0000 0000 BYTE of result stored
       into WKA4

6th comparison

Step (7) 0000 0000 0000 0000 BYTE of dividend in WKA1
       0000 0000 0000 0040 BYTE of divisor in WKA2
       0000 0000 0000 0000 BYTE of remainder in WKA3
       1853 7500 0000 0000 BYTE of result stored
       into WKA4

7th comparison

Step (8) 0000 0000 0000 0000 BYTE of remainder in WKA1
       0000 0000 0000 0040 BYTE of divisor in WKA2
       0000 0000 0018 5375 BYTE of result in WKA3
       0000 0000 0000 0005 BYTE of rounding in WKA4

rounding
Step (9) 0000 0000 0000 0000  BYTE of remainder in WKA1
        0000 0000 0000 0040  BYTE of divisor in WKA2
        0000 0000 0018 5380  BYTE of result in WKA3
        0000 0000 0000 0005  BYTE for rounding in WKA4

The final results are:  
1) the quotient is 185.38  
2) the remainder is 0.

The author would like to demonstrate more clearly as below. First, \(i = 2 + 2 + 1 = 7\), and \(j = 2\).

<table>
<thead>
<tr>
<th>i</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

|   | k 1 2 3 4 5 6 |
|---|--|--|--|--|--|--|
| WKA1 | \(A_j\) 74 341 0215 00150 000300 0000200 |
| WKA2 | \(B_j\) 40 040 0040 00040 000040 0000040 |
| WKA3 | \(R_k\) 34 021 0015 0030 000020 0000000 |
| WKA4 | \(Q_k\) 1 8 5 3 7 5 |

Table 3. A Diagram for Repeated Subtraction in the Division
The purposes of this paper are to

1) review structured programming to determine whether these designs permit easy implementation of the control structures of a flowchartable program designed in a modular manner,

2) outline some of the concepts which the author believes should be incorporated in assembly languages for structured programming by using the control structures in the form of macros, and

3) focus on the design of run-time support routines for arithmetic operations.

Structured design is a set of proposed general program design considerations and techniques used during the programming process. They are used during the phases which involve designing, coding, debugging, and maintaining a program. When they are used properly, structured programming practices can make the above processes proceed faster and less expensively since they tend to result in an end-product that is rather straightforward.
The method employed by top-down design is to break down a system into successively smaller modular subfunctions. In this way, the designer proceeds from general notions or concepts of the system to detailed aspects in a step-by-step manner. This process permits the designer to deal with a controllable amount of complexity, and he can focus attention on a particular facet since its relationship to the rest of the system is fixed and known. The author agrees with Ledgard's concept that using the CALL statement instead of the GO TO statement in a structured program increases program clarity since this method preserves the one-entry, one-exit property. The top-down programming process typified by macros and subroutine calls, in which the main-line code contains only a named reference to code which is refined or spelled out in detail.

Top-down structured programming is the single most important concept used in this design. The benefits of this methodology are described as below:

1) It is far more logical and accurate than traditional approaches.
2) It increases program reliability.
3) It reduces the amount of code which is needed in a particular routine when you adopt subroutine and procedure calls.
4) It makes maintaining the design system easier.

5) It has helped us to improve programmer productivity on an individual and group basis, and the quality of our end-product at the same time.

6) It allows frequent and substantial change in the design systems without upsetting previous work for future plan.

7) It simulates the missing hardware instructions or hardware support used in the compiler.

One way to evaluate structured programming constructs is to examine a system which utilizes them. The interpreter, mentioned in Chapter III, is one such system which was surveyed to determine how heavily each construct was used. Table 4 shows some of the statistics generated from the interpreter source listings.

McGowan and Kelly (1) have noted that programming constructs that are used frequently should be extended in an attempt to eliminate some of the unnecessary uses of those constructs. From Table 4 it appears that the IF-THEN-ELSE is used quite often. From this one would examine the IF construct and then note that by allowing compound conditional expressions, some of the IF statements in the interpreter could have been eliminated.
<table>
<thead>
<tr>
<th>Macro Statement</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF-THEN-ELSE</td>
<td>15%</td>
</tr>
<tr>
<td>BEGIN-END</td>
<td>11%</td>
</tr>
<tr>
<td>EXIT</td>
<td>5%</td>
</tr>
<tr>
<td>CALL</td>
<td>5.3%</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>SUBROUTINE</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>REPEAT</td>
<td>1%</td>
</tr>
<tr>
<td>LOOP</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>CASE</td>
<td>&lt; 0.5%</td>
</tr>
<tr>
<td>EXITLIST</td>
<td>&lt; 0.1%</td>
</tr>
</tbody>
</table>

Table 4. Structured Programming Results from the Interpreter
CHAPTEB BIBLIOGRAPHY

APPENDIX I
STRUCTURED PROGRAMMING
IN ASSEMBLY LANGUAGE

In this appendix the author presents examples with use Chapter IV to develop the top-down structured programs.

DIVIDE          PROCEDURE REGCNT=5,LINKREG=R15

. .

DIVD_GE_DIVS  LOOP

CLHR DIVIDEND_LEN,DIVISOR_LEN IS IT LESS THAN
IF NOT_LOW GT AND EQ
THEN

SUBTRACT_LOOP LOOP

. .

CALL STRING_COMPARE BYTE BY BYTE

*  
LIS WORK_REG,LT GET LT
CLB WORK_REG,CONDITION_CODE
IF EQUAL ,YES
THEN

77
EXIT SUBTRACT_LOOP
ELSE NULL
.
.
CALL SUBTRACT
.
.
END SUBTRACT_LOOP
ELSE
EXIT DIVD_GE_DIVS
.
.
STB DIGIT, BYTE_ZERO(WKA4_LOC) STORE RESULT
*
.
.
END DIVIDE
**** 1 ****
DIVISION FINISH
.
.
"DIVIDE" is the symbol or label name used by another
procedure to refer to this "DIVIDE" procedure in the CALL
macro. There are two CALLs in this procedure. Each
procedure is independent modular unit.

"REGCNT=5" means the contents of the registers 11
through 15 to be saved on entry and restored on exit.
"LINKREG=R15" means the return address of this procedure
to be saved in the register 15.

This procedure does match the algorithm of the division,
described in Section V.4.
I.2 Example of repeat block:

MOVE_STRING    REPEAT COUNT_REG

* FUNCTIONS : THIS ROUTINE WILL MOVE EACH BYTE OF SOURCE
* FIELD TO SPECIFIED LOCATION OF DESTINATION
* FIELD.

* INPUT : 1) R12 TOTAL NUMBER OF BYTES TO MOVE
* 2) R13 ENDING ADDRESS OF SOURCE FIELD
* 3) R14 ENDING ADDRESS OF DESTINATION FIELD

* OUTPUT : 1) R13 UPDATED POINTER TO SOURCE FIELD
* 2) R14 UPDATED POINTER TO DESTINATION FIELD

*-------------------------------------------- -----------------------------

LB TEMP_REG, BYTE_ZERO(SOURCE) GET BYTE
STB TEMP_REG, BYTE_ZERO(DESTINATION) STORE
AIS SOURCE, 1 INCR SOURCE POINTER
AIS DESTINATION, 1 INCR DESTINATION POINTER

END MOVE_STRING ---- END ----

"COUNT_REG" is a register which contains the number of times to be repeated in this loop. It demonstrates a typical documentation in the independent procedure.
1.3 Example of loop block

MOVE_STRING
LOOP
LB TEMP_REG, BYTE_ZERO(SOURCE) GET BYTE
STB TEMP_REG, BYTE_ZERO(DESTINATION) STORE
AIS SOURCE, 1 INCR SOURCE POINTER
AIS DESTINATION, 1 INCR DESTINATION POINTER
*S
SIS COUNT_REG, 1 DECREMENT LOOP-COUNTER
IF ZERO LAST ?
THEN
EXIT MOVE_STRING FINISH AND EXIT
ELSE NULL CONTINUE THIS LOOP
END MOVE_STRING ---- END ----

The LOOP-END figure is a little different from REPEAT-END. It allows a block of codes to be repeatedly executed until one of possible several terminations by use of EXIT, is found.
I.4 Example of subroutine block

MOVE_RESULT_TO_TARGET PROCEDURE REGCNT=3, LINKREG=R15
*

ROUNDING SUBROUTINE SVLINK=R3
*

  THIS SUBROUTINE WILL BE:
*
  1) ROUNDING THE SPECIFIED DIGIT;
*
  2) RETURNING THE TOTAL LENGTH OF DIGITS;
*
  3) GIVING THE SCALE-FACTOR OF THIS NUMBER
*

     IN WORK AREA

LHR WORK_REG,SPECIFIED_LOC
AHI WORK_REG,WORK_AREA_LENGTH GET ADDRESS
LHI ROUND_BYTE,EC'5' GET ROUNDING BYTE
STB ROUND_BYTE,BYTE_ZERO(WORK_REG)
.
.
CALL BYTE_ADD_BYTE EXECUTE ADDITION
.
.
END ROUNDING  *****  *****
.
.
EXECUTE ROUNDDING DO ROUNDDING
END MOVE_RESULT_TO_TARGET ---- END ----
The subroutine figure has been executed, program control will return to "EXECUTE" location in its main program. Inside a subroutine, it allows to "CALL" another procedure, but it needs to save its return address in "SVLINK=R3". Every subroutine block must be contained within some procedure block.

"ROUNDING" is the symbol or label name used by the original procedure in the EXECUTE macro, but not by another procedure to call. A procedure may be invoked by a CALL macro statement that names an entry point of the procedure, which are demonstrated in Appendix I.1 and I.4.
I.5 Example of IF-THEN-ELSE

LH WORK_REG,DATA_TYPE(WORK_AREA)

IF ZERO IS IT CHARACTER-VARIABLE?
THEN
CHARACTER_OP BEGIN
.
.
END CHARACTER_OP ***** 1 *****
ELSE
NUMERIC_OP BEGIN
.
.
END NUMERIC_OP ***** 2 *****

I.6 Example of compound IF-THEN-ELSE

SCAN_NUMERIC REPEAT DIGIT_CNT_LOOP
LB WORK_REG, BYTE_ZERO(DIGIT_ADDR) GET BYTE
IF (CLHI WORK_REG, EC'0'), NL, AND, X
(CLHI WORK_REG, EC'9'), NH)
THEN ,YES, 0 THRU 9
NUMERIC_DIGIT BEGIN
.
.

1
END NUMERIC_DIGIT
ELSE
NOT_DIGIT BEGIN
CLHI WORK_REG,EC' ' IS IT BLANK?
IF NOT_EQUAL ,NOT
THEN
PERIOD_OR_SIGN BEGIN
CLHI WORK_REG,PERIOD IS IT DECIMAL_POINT
IF NOT_EQUAL , IT IS A SIGN
THEN
YES_SIGN BEGIN
IF ((CLHI WORK_REG,PLUS),EQ,OR,
(CLHI WORK_REG,MINUS),EQ) X
THEN
STB WORK_REG,SIGN(WORK_AREA) STORE SIGN
ELSE
SET_ERROR BEGIN
.
.
END SET_ERROR
ELSE
END YES_SIGN
ELSE
END
This example matches Figure 1b. It provides for the testing of logical combinations of conditions by the use of "AND" or "OR".

In example of Appendix I.5, the IF statement provides the common conditional branch and is usually used with a simple comparison expression following the word IF.
I.7 Example of case block

ERROR BEGIN

* * * * * START CASTE * * * *

CASE R10

* 

#CASE1 NULL NORMAL
#CASE2 EXIT ERROR,2 I/O ERROR
#CASE3 EXIT ERROR,2 OPERATOR ABORT
#CASE4 EXIT ERROR,1 PARAMETER ERROR
#CASE5 EXIT ERROR,1 END OF FILE
#CASE6 EXIT ERROR,2 FILE NOT FOUND
#CASE7 EXIT ERROR,2 MACRO ERROR
#CASE8 EXIT ERROR,2 READ AFTER EOF
#CASE9 EXIT ERROR,1 FILE SIZE EXCEEDED
#CASE10 EXIT ERROR,2 SOF AFTER BACK SPACE

ELSE

EXIT ERROR,2 ANOTHER ERROR

.

.

EXITLIST ERROR_CODE

SET_UP_TRAP BEGIN

.

.

.

.
A begin block is delimited by a BEGIN statement and an associated END statement. Begin blocks are executed in the normal flow of a program, either sequentially or as a result of a IF-THEN-ELSE statement or an EXITLIST transfer.

The EXITLIST statement indicates the beginning of a list of statements, one of which may be executed before executing a block. In this example of CASE figure the "SET_UP_ERROR" begin block is starting, it must find an "END" matching statement to terminate its function.

In case figure "R10", i.e. an index integer, will be checked for range before execution and if it is out of range, control will proceed to the ELSE statement.
**Note:** The value of data type is 0, which represents a character.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Length</th>
<th>Data Type</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN(3)'INIT'(0)</td>
<td>3</td>
<td>1</td>
<td>Variable</td>
<td>Fixed</td>
</tr>
<tr>
<td>N(6)'INIT'(5)</td>
<td>6-6</td>
<td>2</td>
<td>Right</td>
<td>Length</td>
</tr>
<tr>
<td>VC(7)'INIT'(1,7)</td>
<td>7</td>
<td>0</td>
<td>Character</td>
<td>Right</td>
</tr>
<tr>
<td>C(3)'INIT'(1,3)</td>
<td>3</td>
<td>0</td>
<td>Fixed</td>
<td>Length</td>
</tr>
</tbody>
</table>

Example of the declaration:
BIBLIOGRAPHY

Books


Articles


Miller, E. F. Jr and G. E. Lindamood, "Structured


