

1971

A programming system for the simulation of digital machines

Bruce A. Pumplin
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Electrical and Electronics Commons](#)

Recommended Citation

Pumplin, Bruce A., "A programming system for the simulation of digital machines " (1971). *Retrospective Theses and Dissertations*. 4501.
<https://lib.dr.iastate.edu/rtd/4501>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

72-5249

PUMPLIN, Bruce A., 1941-
A PROGRAMMING SYSTEM FOR THE SIMULATION OF
DIGITAL MACHINES.

Iowa State University, Ph.D., 1971
Engineering, electrical

University Microfilms, A XEROX Company, Ann Arbor, Michigan

A programming system for the simulation of digital machines

by

Bruce A. Pumplin

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Electrical Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University

Ames, Iowa

1971

PLEASE NOTE:

Some Pages have indistinct
print. Filmed as received.

UNIVERSITY MICROFILMS

TABLE OF CONTENTS

	Page
INTRODUCTION	1
Motivation	1
Relation to Other Work	3
THE BAPSIM LANGUAGE	6
Introduction and Background	6
Language Requirements	8
The Meta-Language	10
The BAPSIM Vocabulary	15
Identifiers	16
Integers	16
Operators	17
Delimiters	21
Declarators	22
Program Structure	22
Program Statements	26
Declaration statements	26
Simple statements	29
A Sample Program	36
THE BAPSIM TRANSLATOR	49
Introduction	49
Syntactic Analysis Techniques	52

The Method of Syntactic Functions	54
Implementation Details	57
Primitive procedures	59
Non-primitive procedures	62
The Simulator	64
Primitive procedures	65
Code generation	66
Simulator I/O	68
Primitive operations	70
CONCLUSIONS	77
BIBLIOGRAPHY	82
APPENDICES	86
Appendix 1. The BAPSIM Syntax	87
Appendix 2. A Sample Program	90
Appendix 3. The Syntax Analyzer	93
Appendix 4. The BAPSIM Translator	113
Appendix 5. Simulator Flow Chart	142
Appendix 6. DCLS	146
Appendix 7. ENDCODE	149
Appendix 8. PRIMPROC	151

INTRODUCTION

Motivation

The purpose of the work reported on here was to develop a programming system for the simulation of digital machines. The digital machines of interest in this work comprise the class of stored program, automatic digital computers within the much more extensive universe of digital machines in general. The current interest in, and the importance of, this subset of digital machines served as the justification for restricting the focus of attention to one specific area.

The simulation system discussed here is made up of two equally important elements. The first is a machine-readable, formal notation for specifying the structure and the behavior of digital processors at the register-transfer level. The second is a translator or language processor for converting a machine description in the above notation into a simulator which is capable of "executing" programs written in the machine language of the processor being simulated.

The computer description language developed here serves two purposes. First, the language is capable of specifying in a precise manner the structure of the digital machine being studied. Compared to existing informal notations for describing machine structure, precision of description is

achieved at the cost of a rigid language structure. However, since the language structure intentionally reflects the structure of most present day digital processors, the result is a computer description language which is precise and at the same time easy to use. In addition, the rigid structure of the language developed here makes it possible to machine-read the computer description and opens the door to a wide variety of automatic calculations using the machine description as input.

Second, the language is capable of specifying the behavioral characteristics of the digital machine being studied. It is this second capability of the language which makes the simulation of machine behavior possible. If machine descriptions in the language were capable of specifying only machine structure their usefulness would be restricted. They could be used for formal documentation of the machine design or they could serve as input to an automated procedure for developing interconnection diagrams and/or Boolean equations. But machine descriptions that omit specifying the behavior of the machine are incapable of serving as input data to an automated simulation system.

The translator or language processor developed here serves two purposes. First, from the user's viewpoint, it converts the machine description into a machine simulator. And it is the machine simulator which allows the user to

observe and study machine behavior. Formally, the simulator constructed from the machine description is capable of mapping an initial machine state onto a sequence of output states. When the simulator is creating the output sequence it is simulating the behavior of the digital machine.

Second, the translator provides the precise semantic meaning for the computer description language. The form or syntax of the language is specified by the language definition. The language definition by itself conveys no meaning. It is only when one knows what meaning the language processor attaches to any given statement in the language that one knows what the statement means.

Together the computer description language and its associated language processor make up a programming system for the simulation of digital machines. Such a simulation system may be used (1) by the computer designer as a design tool to assist him in the analysis and evaluation of alternative machine designs or, (2) by the computer instructor as an educational tool to assist him in the illustration of fundamental computer structural and behavioral concepts.

Relation to Other Work

The language discussed below leans heavily on the pioneering work of Chu (11) and the more recent work of Bell

and Newell (04). The concept of a block-structured language, after the work of Naur (27), also influenced the language. The work of Bartee, Lebow, and Reed (01) was influential in determining the machine level at which machine behavior was to be described. The language which emerged from the work done here is thought to take advantage of the best features of previous efforts in this area without incorporating their various inadequacies.

The translator discussed below owes no such debts to previous work in the area of language processors. Those authors recognizing the usefulness of developing a machine simulation system stop short of discussing processors for their computer description languages. Other authors do not even discuss the possibility of developing a simulator. The technique implemented by the translator here is a general technique applicable to all language processors. Bolliet (06) has applied the method of syntactic functions, in an incomplete manner, to an ALGOL-like higher level programming language.

The first chapter below describes the syntactic details of the computer description language developed here. The second chapter below supplies the semantic meaning of the language and discusses the implementation details of the language translator. The third chapter below discusses the results which were obtained and the implications which follow

from these results. Various program listings, the formal syntactic definition of the language, and a sample machine description will be found in the appendices.

THE BAPSIM LANGUAGE

Introduction and Background

The first goal of the research reported on here was to define a machine-readable formal notation for specifying the structure and behavior of digital processors at the register-transfer level. Although no such notation has been accepted as a standard, the desirability of a convenient, concise method for describing the details of digital machines has been recognized for some time.

Bartee, Lebow, and Reed (01) present a rigorous treatment of digital machines by considering their operation at the register-transfer level. This approach to machine description recognizes that (1) machine structure may be defined in terms of registers and data paths, and (2) machine behavior may be defined in terms of information flow between registers along the existing data paths. Rather informally, the authors of this standard textbook develop a printable symbology which they use to describe certain fundamental concepts of computer behavior. Loaded with subscripts, superscripts, and Greek characters, however, their printable notation is far from machine-readable.

Gorman and Anderson (19) present the conceptual details of an experimental programming system to be used by the logic

designer as an aid in the development of logic equations for digital computers. In a companion effort, Proctor (29) discusses "a system descriptive language" used to provide the description of the digital computer under investigation. Along with his co-workers, Proctor must be given credit for recognizing the usefulness of developing a formal language for machine description which is machine-readable.

Schlaepfi (31) provides an informal description of a proposed "behavioral hardware language" for describing machine logic, timing and sequencing and suggests possible uses for such a language. In fact, he suggests the possibility of constructing a processor for such a language which would automatically translate the description of an object machine into a simulator capable of executing sample programs written in the language of the object machine. Just such a language processor is the second goal of the research reported on here.

Chu (11) proposes modelling "a computer design language" along the lines of a high level programming language and discusses the advantages of such a technique. The high level programming language he selects is ALGOL. Chu also recognizes the possibility of developing a translator for use with such a language as a step toward the simulation of the object machine on an existing machine.

Bell and Newell (04) press the case for development of a

computer descriptive language and present the details of their efforts in this area in their book (03). Their notation achieves the goals of precision and flexibility of description necessary in such a language, but lacks the virtues of lucidity and convenience. Further, as their notational system now exists, it is not machine-readable. In passing, these authors note the usefulness of developing a simulator for executing object machine programs.

Language Requirements

The purpose to be fulfilled by the language being defined is sufficient to dictate a number of general requirements the language must meet.

First, the language must be easy to learn. This requirement is best met by making the language as close as possible to natural language or, next best, making it as close as possible to an existing programming language. The additional requirements that the language be unambiguous, concise, and precise exclude the possibility of using a natural language model for a computer design language. However, making the computer design language mirror, wherever possible, the constructs and concepts of existing high level programming languages will contribute significantly to the "naturalness" of the language.

Second, the requirement that the language be convenient to use can be met by ensuring that the elements from which a machine description is constructed correspond directly and obviously to the structural and behavioral elements of the object machine being described. For example, in describing digital machines at the register-transfer level, there must exist an element of the design language which directly models the hardware registers of the object machine.

Third, in order that the language be concise, it must suppress all details of machine description and machine behavior which are of no interest or are of interest only in so far as their functions must be defined. For example, at the register-transfer level, the details of the instruction decoding operation are usually of no interest. In the case of elementary machines, a simple mapping from operation code to operation microsequence is sufficient.

Fourth, one of the most important benefits to be derived by the computer programmer from using a high level programming language is the ability to name and refer to symbolically, objects defined in terms of primitive language constructs. By requiring the language used in describing machine operation to have the same sort of hierarchical structure, the same benefits of lucidity and flexibility may be realized.

The above requirements have guided the development of

the BAPSIM computer description language described in detail below. The extent to which these requirements have been met will be left for the reader to determine for himself.

The Meta-Language

The syntax of the BAPSIM computer description language will be described in what follows with the aid of a modified version of the meta-notation developed by Brooker and Morris(08). The use of this meta-notation will be explained first by a series of examples. Consider the meta-definition

```
DIGIT = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

The sequence of five characters on the left hand side of the equals sign is a meta-linguistic variable used to denote any occurrence of the syntactic unit of the same name. The equals sign is a meta-linguistic character that is to be read as "is defined to be". Quote marks are meta-linguistic characters used to delimit character strings which occur literally. The vertical slash is a meta-linguistic connective which means "or". In words, this definition states that "a digit is defined to be the literal occurrence of a character zero, or a literal occurrence of a character one, or a literal occurrence of a character two, ..., or a literal occurrence of a character nine".

Similarly, the syntactic unit "letter" is defined:

```
LETTER = 'A'|'B'|...|'Y'|'Z'
```

In the possibly more familiar Backus Normal meta-notation, the definition of a digit would appear as:

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

The differences between the meta-notation to be used here and the meta-notation developed in the ALGOL 60 report are based on a desire for clarity and simplicity in the meta-notation. In the definition of any usefully large language, meta-linguistic variables occur much more frequently than literal character strings. By delimiting character strings rather than meta-linguistic variables the language definition is shorter and easier to read.

To avoid ambiguity in specifying the juxtaposition of syntactic units it is necessary (and sufficient) to require that the un-delimited meta-linguistic variables used to denote the syntactic units of the language be unbroken character strings. Use of the break character, '_', instead of the blank results in an unambiguous, easy-to-read notation. An example will clarify this last point.

In Backus Normal form, <unsigned integer> is a valid meta-linguistic variable. Merely dropping the delimiting brackets results in an ambiguous construction if it appears

on the right hand side of a syntactic definition. Does it refer to the single syntactic unit "unsigned integer" or does it refer to the two syntactic units "unsigned" and "integer"? In the notation used here, if it does indeed refer to the single syntactic unit, the embedded blank must be replaced by the break character, i.e., UNSIGNED_INTEGER. In the present notation UNSIGNED INTEGER will denote the occurrence of any member of the syntactic unit "unsigned" followed by the occurrence of any member of the syntactic unit "integer".

The meta-notation used here replaces the recursive definitions occurring so frequently in the ALGOL 60 report with iterative definitions. The full implications of this change will be discussed later when the syntax-directed (10) BAPSIM translator is described.

In Backus Normal form the definition of an "identifier" would be:

```
<identifier> ::= <letter>|<identifier><letter>|
               <identifier><digit>
```

This definition is an excellent example of a definition which is left recursive.

In the meta-notation used here, the definition of an "identifier" is:

```
IDENTIFIER = LETTER $ (LETTER|DIGIT)
```

The dollar sign introduced in the last definition is a meta-linguistic character that is to be read as "zero or more occurrences of". In words, this non-recursive definition of an "identifier" states that "an identifier is defined to be a letter followed by zero or more occurrences of a letter or a digit, in any order". Similarly, the syntactic unit "integer" is defined to be:

$$\text{INTEGER} = \text{DIGIT } \$ (\text{DIGIT})$$

In what follows, the syntactic units "identifier" and "integer" will be considered primitive units or terminal symbols and will be denoted by .ID and .INT respectively. As the BAPSIM language is defined, identifiers and integers may be of any length. The BAPSIM translator, which constructs from the BAPSIM machine description a simulator for the object machine, however, requires that identifiers be limited to character strings of length eight or less and integers be limited to character strings of length five or less.

The special meta-linguistic symbol .EMPTY will be used to indicate the optional occurrence of a member of a syntactic unit. Strictly speaking, this special symbol refers to an occurrence of the null string of characters. For example, the meta-definition:

$$A = X (Y\{\text{.EMPTY}\})$$

states that "an A is defined to be an X followed by either a Y or the null string". In other words, "an A is either an X or an XY".

The BAPSIM Vocabulary

The BAPSIM computer description language is built up from the basic symbols shown in TABLE 1. As explained in the previous section, the quote marks and the vertical slash which appear in TABLE 1 are meta-linguistic characters and are not a part of the BAPSIM language.

TABLE 1. THE BAPSIM VOCABULARY

Identifiers	LETTER \$ (LETTER DIGIT)
Integers	DIGIT \$ (DIGIT)
Operators	
Arithmetic	' .ADD.' '.SUB.'
Logical	' =.' '*.' '+.' '.NOT.' '.AND.' ' .OR.' '.XOR.'
Functional	' .SHL.' '.SHR.' '.CIRL.' '.CIRR.'
Relational	' .EQ.' '.NE.' '.LT.' '.GT.' ' .LE.' '.GE.'
Sequential	' <-' ':=' 'GO TO' 'IF' 'THEN' 'DO'
Delimiters	' (' ')' ',' '<' '>' '-' 'END' ' SIMULATION' ' DECLARE' ' FETCH' ' DECODE' ' EXECUTE'
Declarators	' REGISTER' ' SUBREGISTER' ' MEMORY' ' INITIALIZE' ' MONITOR' ' TERMINAL' ' OPERATION'

Identifiers

Identifiers are names given to the structural and behavioral elements of the object machine being described. As such, identifiers are used to name the four kinds of machine structural elements of interest -- registers, subregisters, memories, and terminals -- and the one behavioral element of interest -- the operation. In general, operations are nothing more than named collections of primitive machine operations. The capability for the user to name, define, and refer to collections of primitive machine operations as he wishes gives the language the desired hierarchical structure. This provision also enables the suppression of detail essential to a clear machine description.

Identifiers naming structural elements of the machine must be declared before they are used. Identifiers naming machine operations corresponding to the object machine's instruction set must not be explicitly declared; contextual declarations are sufficient in these cases. All identifiers naming user-defined collections of primitive machine operations must be explicitly declared.

Integers

The only numbers that may appear in a BAPSIM computer

description are unsigned decimal integers of length five or less. These integers are used in the declaration of registers, subregisters, and memories; in referring to individual bits of a register or subregister or to individual words of a memory; and in referring to binary bit patterns contained in a register, subregister, or memory word.

Operators

Five types of operators are recognized as primitive symbols of the BAPSIM language: arithmetic, logical, functional, relational, and sequential. Some of these operators are unary -- they are associated with one identifier operand -- and some are binary -- they are associated with two identifier operands or one identifier operand and one integer operand -- and some can be either unary or binary. In the latter case, the exact operation denoted by the operator is determined from the context in which the operator appears.

The two arithmetic operators are binary in nature and correspond to the usual arithmetic operations of addition and subtraction.

The single-character logical operators shown in TABLE 1 ($\neg, *, +$) may appear only in terminal statements and correspond naturally to the logical operations of complementation, logical product, and logical sum.

The multi-character logical operators `.NOT.` , `.AND.` , and `.OR.` may appear in both IF-statements and transfer statements; the logical operator denoting the exclusive-or operation, `.XOR.` , may appear only in transfer statements. The logical operator denoting the logical complement operation, `.NOT.` , is a unary operator; the operator denoting the exclusive-or operation is a binary operator. The remaining two multi-character logical operators, `.AND.` and `.OR.` , may be either unary or binary operators depending on the context in which they appear. The result of applying the unary-and operator to a register or subregister operand is a single bit whose value is '1' if all of the bits of the operand are '1' and '0' otherwise. The result of applying the unary-or operator to a register or subregister operand is a single bit whose value is '1' if any of the bits of the operand are '1' and '0' if all of the bits of the operand are '0'. The other operators are applied bit-by-bit to the operand(s).

The functional operators appearing in TABLE 1 represent a class of operator which does not appear in the definition of any existing high level programming language. They have been included in the BAPSIM computer description language because they correspond directly and obviously to elementary machine operations occurring at the register-transfer level.

The functional operators shift left, `.SHL.` , and shift

right, `.SHR.`, may be either unary or binary operators. As defined here, the shift left operation shifts the bits of a register or subregister operand one bit to the left and inserts a '0' at the right end; the shift right operation shifts the bits of a register or subregister operand one bit to the right and inserts a '0' at the left end. Other shift operations, such as shifting and retaining the left- or right-most bit, are recognized as valid fundamental machine operations at the register-transfer level but have not been implemented in the current version of BAPSIM.

When appearing as unary operators in transfer statements the associated operand is an identifier naming the register or subregister which is to undergo the one-bit shift operation. When appearing as binary operators in transfer statements the first operand is an identifier as in the case of a unary operator and the second operand is an integer specifying how many times the one-bit shift operation is to be performed. Multiple shifts are thus provided for.

The functional operators circulate left, `.CIRL.`, and circulate right, `.CIRR.`, may appear only as unary operators in transfer statements. The circulate operations are nothing more than end-around one-bit shift operations. A reasonable extension to the current version of the BAPSIM language would be to allow the two circulate operators to also appear as binary operators in transfer statements as in the case of the

two shift operators, thereby allowing multiple circulate operations. The infrequent occurrence of such multiple circulate operations excluded this provision in the current version of the BAPSIM language.

The relational operators appearing in TABLE 1 may appear only in IF-statements and carry obvious meanings. Those readers familiar with FORTRAN will immediately recognize the source of the particular notation used for representing these operators.

The sequential operator '<-' is used only in transfer statements and serves to denote a register-transfer operation. This operator may be read as "is replaced by" or "becomes". The intuitive appeal of the chosen notation should be obvious. The sequential operator ':=' is used only in terminal statements and serves to denote the familiar assignment operation of high level programming languages. By selecting different notations for the register-transfer operation and the terminal-assignment operation the essential difference between the two operations is emphasized. The remaining four sequential operators are borrowed directly from existing high level programming languages. The DO operator, however, has a slightly unconventional, though quite natural, meaning: DO followed by an identifier naming an explicitly defined operation serves to call for the operation named. Thus, DO in BAPSIM is equivalent to CALL

in FORTRAN or PL/1. The DO of FORTRAN or PL/1 has no equivalent in BAPSIM.

Delimiters

The single-character delimiters shown in TABLE 1 are used in a standard manner. Details of their use will be explained below. The multiple character delimiters are used to implement the block structure of programs (i.e., machine descriptions) written in BAPSIM. As might be expected from the number of program structure delimiters in the language, legal BAPSIM programs have a rigid structure. However, far from being a hinderance to machine description, this rigid structure required of BAPSIM programs actually facilitates writing, reading, and understanding machine descriptions. The reason for this is that the structure of BAPSIM programs directly reflects the structural and behavioral elements of the machine being described. The structural elements of a machine -- the registers, the subregisters, the memories, and the terminals -- are all specified in a program declare block. User defined operations, representing machine behavioral elements, also must be declared in the declare block. Specification of machine elements for performing the basic fetch, decode, and execute operations common to all present day digital computers also must be done within program blocks corresponding to these basic operations. The END

delimiter is used to signal the end of a basic program block.

Declarators

The first five declarators shown in TABLE 1 correspond to the five types of machine elements; the first four are structural elements and the fifth is a behavioral element. The final two declarators, INITIALIZE and MONITOR, serve to define an initiation procedure and an output procedure, respectively, for the simulator constructed by the BAPSIM translator. These two declarators supplement the actual machine description and, for this reason, must not appear in the same block as the other declarators which are an integral part of any machine description.

Program Structure

The BAPSIM vocabulary described in the preceding section provides the raw materials from which BAPSIM programs are constructed. This section will discuss the syntax of machine descriptions written in BAPSIM. The meta-language discussed previously will be used to formally define the syntax of BAPSIM programs.

A BAPSIM program consists of one outer block which contains three inner blocks. Here a block is taken to be a delimited sequence of statements that constitutes a

functional section of a program.

```
PROGRAM = SIMULATION_BLOCK
```

```
SIMULATION_BLOCK = 'SIMULATION' STRUCTURE_BLOCK
                   BEHAVIOR_BLOCK IN_OUT_BLOCK
                   'END SIMULATION'
```

The structure block of a BAPSIM program contains all of the program declarations and thus reflects the structural characteristics of the machine being described. In addition, user-defined operations must be declared in the structure block. Even though these operations actually reflect certain behavioral characteristics of the machine being described, it was deemed appropriate to group all of the program declares at the beginning of each program.

```
STRUCTURE_BLOCK = 'DECLARE' DECLARATION_STMT
                  $ ( ';' DECLARATION_STMT )
                  'END DECLARE'
```

The behavior block of a BAPSIM program is, in turn, made up of three other blocks, each of which reflects a specific behavioral aspect of the machine being described.

```
BEHAVIOR_BLOCK = FETCH_BLOCK DECODE_BLOCK EXECUTE_BLOCK
```

The fetch block of a BAPSIM program contains a sequence

of statements which specifies the fetch operation of the digital machine being described. Operand address calculation and next-instruction address calculation sequences may also be specified in the fetch block.

```
FETCH_BLOCK = 'FETCH' SIMPLE_SEQUENCE 'END FETCH'
```

The decode block of a BAPSIM program contains a sequence of statements which specifies the instruction decoding operation of the object machine. In the description of simple machines the decode sequence need be nothing more than a series of IF-statements which serve to map the value contained in a register or subregister into the name of the machine operation being decoded. In the description of more complex machines, specifying the decoding operation may require the use of other types of statements as well.

```
DECODE_BLOCK = 'DECODE' SIMPLE_SEQUENCE 'END DECODE'
```

The execute block of a BAPSIM program contains a sequence of statements which specifies the instruction interpretation operations of the object machine being described. Each machine instruction must be named -- so that it may be referred to in the decode block -- and the micro-operations required to "execute" the machine instruction must be specified. In most cases these micro-operations are merely the elementary register-transfer operations. Each of these

elementary operations is specified by a simple statement in the BAPSIM language. These simple statements were defined to reflect the elementary machine operations.

```
EXECUTE_BLOCK = 'EXECUTE'  COMPOUND_STMT
                $ ( ';' COMPOUND_STMT )
                'END EXECUTE'
```

It should be noted here that the syntactic unit COMPOUND_STMT appearing in the definition of an EXECUTE_BLOCK is nothing more nor less than a labeled simple sequence, the label serving to name the operation specified by the simple sequence.

The input/output block of a BAPSIM program contains the statements required to specify the initiation procedure and the output procedure required for the simulator constructed by the BAPSIM translator. Since these statements are not a part of the actual machine description, but rather a supplement to it, they must appear in a separate block in the BAPSIM machine description.

```
IN_OUT_BLOCK = INITIALIZATION_STMT  MONITOR_STMT
```

All of the various blocks (i.e., programs segments) making up a BAPSIM program have now been defined and described. The purpose of this section has been to explain the block structure of a BAPSIM machine description and to

highlight the manner in which this block structure is intended to reflect the structure and behavior of the object machine being described.

Program Statements

The statements which comprise BAPSIM machine descriptions fall into two broad categories: declaration statements, which serve to specify the machine structure, and simple statements, which are used to specify machine behavior.

Declaration statements

The set of program statements referred to collectively as declaration statements is made up of five members: register declarations, subregister declarations, memory declarations, terminal declarations, and operation declarations. Each of these statement types is defined, using the meta-notation described earlier, in Appendix 1 which shows the complete syntactic definition of the BAPSIM language. In this section the various program declaration statement types will be illustrated by example.

Register declaration statements are required to specify the single- or multi-bit hardware registers of the object machine. The machine registers are named and their lengths are specified as shown below:

```
REGISTER      A(0-31),Q(0-31),OV;
```

This register declaration statement specifies two 32-bit registers, A and Q, and one 1-bit register, OV. In the case of multi-bit registers, bit positions are assumed to be numbered zero through n-1, where n is the length of the register in bits, starting from the left. This convention is required by the BAPSIM translator in its construction of the object machine simulator and is not a part of the BAPSIM language.

Subregister declaration statements provide the facility for naming, and hence referring to, contiguous portions of previously declared registers. Assuming register A has previously been declared as in the above example, we might have:

```
SUBREGISTER   A(OP)=A(0-7),A(ADDRESS)=A(8-31);
```

This subregister declaration statement associates with the first eight bits of register A the name OP and with the final twenty-four bits of register A the name ADDRESS. If register A may also contain sign-magnitude fixed-point data, we may want to include the following additional subregister declaration:

```
SUBREGISTER   A(SIGN)=A(0),A(MAG)=A(1-31);
```

This statement associates the name SIGN with the first (i.e., left-most) bit of register A and the name MAG with the final

(i.e., right-most) thirty-one bits of register A.

Memory declaration statements provide a means for naming and defining the size of the memories of the object machine.

```
MEMORY      M(MAR)=M(0-127,0-31);
```

This memory declaration statement asserts that the object machine being described has a single 128-word memory. The memory word length is thirty-two bits and the memory is accessed by the memory address register having the name MAR.

Terminal declaration statements name and define the number of terminals to be used in the description of object machine behavior. As Chu (11) points out, having this set of special identifiers, used to designate signals at critical circuit points, is convenient in some situations and necessary in others. The use of terminals in a machine description will be illustrated in the section below on terminal statements. A sample terminal declaration follows:

```
TERMINAL    K(0-7),SUM,T5,CARRY(0-14);
```

This statement associates the name K with a group of eight terminals, the names SUM and T5 with single terminals, and the name CARRY with a group of fifteen terminals.

Operation declaration statements are used to declare the presence of user-defined collections of primitive machine operations in the behavioral description of the object machine.

Such declarations are necessary in order that the BAPSIM translator be able to differentiate between user-defined operations and the operations corresponding to the object machine instruction set. The object machine simulator constructed by the BAPSIM translator treats these two types of operations in different ways. An alternative to requiring the explicit declaration of user-defined operations would be to adopt some sort of naming convention to distinguish between the two types of operations, say require all user-defined operations, and only user-defined operations, to begin with the letter Z. The associated reduction in naming flexibility, from the viewpoint of the programmer, was considered to be less desirable than requiring explicit declarations.

```
OPERATION      MULT,ROOT,ADDRESS;
```

This operation declaration statement alerts the BAPSIM translator to the existence of three user-defined operations (MULT,ROOT,ADDRESS) in the description of the object machine.

Simple statements

The set of program statements referred to collectively as simple statements is made up of five members: DO-statements, IF-statements, GO_TO-statements, transfer statements, and terminal statements. Each of these statement

types is defined in Appendix 1. In this section the various simple statement types will be illustrated by example.

DO-statements serve to invoke the specified user-defined operation. When executed by the simulator for the object machine, DO-statements in BAPSIM are equivalent to CALL-statements in FORTRAN or PL/1. For example, the statement

```
DO ROOT
```

calls for the execution of the statements appearing in the definition of the user-defined operation ROOT. After the operation called for by the DO-statement is complete, "control" returns to a point immediately following the DO-statement. Any identifier appearing in a BAPSIM program immediately following the keyword DO must have previously been declared to be a user-defined operation.

IF-statements serve to specify the conditional occurrence of an action. For example, the statement

```
IF ( OP .EQ. 1 ) THEN DO ADD
```

specifies that the user-defined operation ADD will be carried out if and only if the register, subregister, or terminal named OP has a value equal to 1. In general, the action following the keyword THEN will be carried out if and only if the condition following the keyword IF is true. Readers familiar with PL/1 or ALGOL should note that this is the only

conditional construction allowed in BAPSIM. The construction IF...THEN...ELSE, legal in other high level programming languages, is not a valid BAPSIM construction.

GO_TO-statements serve to specify unconditional transfers of control within a BAPSIM program. The identifier appearing in the GO_TO-statement must appear elsewhere in the machine description as a statement label. The meaning and use of GO_TO-statements in BAPSIM are identical to the meaning and use of GO TO-statements in other high level programming languages.

Transfer statements are the heart of the BAPSIM language since they serve to specify the object machine register-transfer operations. These operations, being nothing more nor less than the primitive machine operations, completely define the behavioral characteristics of the machine being described. Transfer statements specify the information transfer from one register to another as well as the logical operation, if any, performed during the transfer. For example, a simple direct transfer of information between two registers is specified in BAPSIM as:

```
A <- Q
```

This statement calls for the transfer of the contents of register Q to register A. Examples of more complex transfer statements and word descriptions of their meaning follow.

$$X \leftarrow M(\text{MAR})$$

This statement represents a memory fetch operation. It calls for the transfer of a word, specified by the contents of the memory address register MAR, out of memory M into register X.

$$M(\text{MAR}) \leftarrow X$$

This statement is a memory store operation whose meaning is exactly analogous to the preceding memory fetch operation.

$$A \leftarrow \text{.SHR. } X$$

This statement specifies a shift and transfer operation. The contents of register X are transferred to register A shifted one bit to the right. The contents of register X are unaltered by this operation.

$$A \leftarrow A \text{ .ADD. } X$$

This statement specifies an addition operation. The contents of register A are added to the contents of register X -- in a strict binary sense -- and the result is returned to register A. The operation is the basic machine addition operation and implies the existence of an adder of some sort in the object machine. The logical details of the adder are not specified by this operation and are of no concern when viewing the object machine on the register-transfer level.

Terminal statements specify the signals at critical circuit points called terminals. Terminals may be thought of as 1-bit registers whose value is specified as some logical function of other 1-bit registers, subregisters or terminals. For example, the output-carry terminal, *OV*, of a full adder may be considered a terminal whose value is specified by the terminal statement

$$OV := K(0) * (\neg A(0)) * (\neg R(0)) + (\neg K(0)) * A(0) * R(0)$$

Use of the sequential operator `:=` serves to distinguish terminal statements from transfer statements and highlights the fact that the operations specified by these two types of statements are different in essence. The only logical operators which may appear in terminal statements are the logical product operator `*`, the logical sum operator `+`, and the logical complement operator `¬`. These three operators are sufficient for specifying any logical function which may be associated with a terminal. It should be noted that the logical operators which appear in terminal statements are not the same operators which appear in transfer statements. The reason for this distinction is that terminal statements may be much longer than transfer statements and single-character operators make the longer statements easier to read (and write). The example given above has no equivalent transfer statement; a terminal statement is necessary to specify the

operation. There are cases where a terminal statement is merely more convenient to use than a transfer statement. For example,

```
A(0) <- A(0) .AND. R(0)
```

and

```
A(0) := A(0)*R(0)
```

are equivalent.

Simple statements may optionally specify timing information for use by the simulator in simulating machine behavior. If timing data is to be supplied, the format is:

```
SIMPLE_STMT <TIME>
```

where TIME is an integer which specifies the time, in relative units, required to perform the operation specified by the simple statement. For example, if a memory access operation takes 1.5 microseconds in a certain machine, the simple statement describing this operation might appear as:

```
X <- MEM(MAR) <15>
```

In this case, the basic time unit is clearly one-tenth of a microsecond and the memory access operation is performed in fifteen of these basic time units.

As a further example, if the machine being simulated

employs an adder (of some sort) which is capable of adding the contents of two registers and returning the result to one of the two registers in 2.0 microseconds, the simple statement describing the machine's addition operation might be:

```
A <- A .ADD. X <4>
```

In this case, the basic time unit was chosen to be one-half of a microsecond.

The choice of the basic time unit is left to the BAPSIM programmer in order to provide him with the ability to describe machines of widely varying speed. The only restriction on the choice of the basic time unit is that all machine operations must take place in time intervals which are integral multiples of the basic time unit.

If timing information is specified in the object machine description, the simulator keeps track of the (simulated) time required to perform all of the operations in each of the (simulated) fetch-decode-execute cycles. At the end of each cycle the accumulated time is printed out along with the machine state. This feature, then, provides information about the time required for the object machine to execute a given program.

In the case of synchronous machines, where each and every fetch-decode-execute cycle takes place in the same amount of time, the timing information provided is not too

useful since the total time is merely the product of the time per cycle and the number of cycles executed. In the case of asynchronous machines, where the time required for each cycle is a function of the operations performed during the cycle, the timing information provided automatically by the simulation run would be difficult to obtain in any other way.

A Sample Program

The purpose of this section is to further clarify, by means of an example, the structure of BAPSIM programs and the syntax of BAPSIM statements.

The digital computer described in this section (referred to in what follows as TELCOMP) does not exist in hardware. Rather it is a hypothetical machine designed to be used as a pedagogical tool for illustrating fundamental concepts common to all digital computers. For precisely this reason, it is also an excellent vehicle for illustrating the structure of BAPSIM machine descriptions and the syntax of BAPSIM program statements.

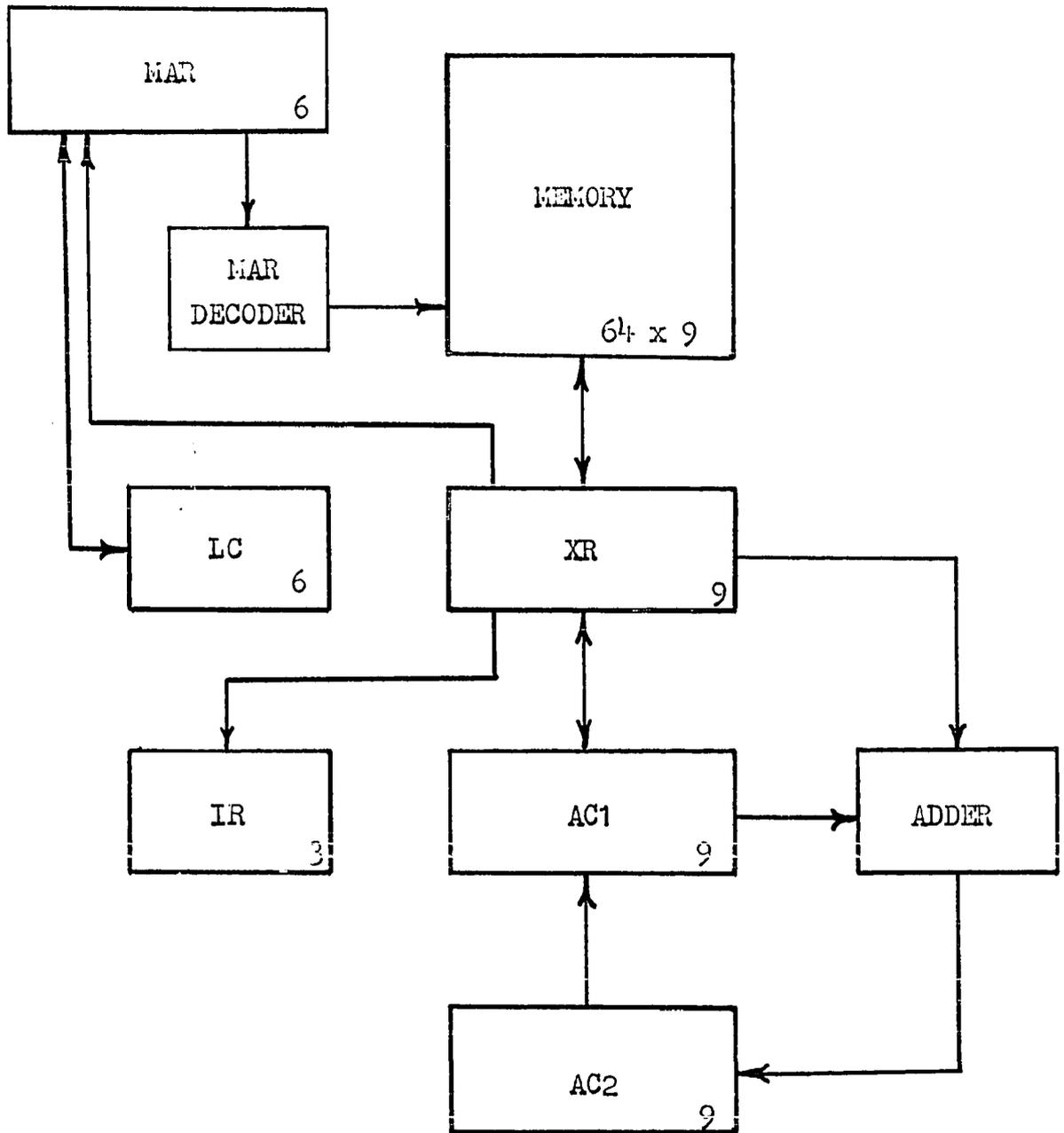


FIGURE 1. THE TELCOMP COMPUTER.

FIGURE 1 is a block diagram of the TELCOMP computer illustrating in an informal manner the structural characteristics of the machine. The relationship between this pictorial description of the machine and the formal BAPSIM machine description will be discussed below. In referring to the BAPSIM program (Appendix 2) reference will be made to the identification information appearing as 8-character strings to the right of the figure. This information is not a part of the BAPSIM language and would not appear in an actual BAPSIM program.

The BAPSIM translator, which constructs from the BAPSIM machine description a simulator for the object machine, recognizes all program cards with an asterisk '*' in column 1 as comment cards. Comment cards are merely listed by the BAPSIM translator and hence are not a part of the formal machine description. They allow the programmer to supplement his formal machine description with English language explanations and identification information.

The formal machine description begins on line 1030. The literal occurrence of SIMULATION signals the beginning of a simulation block and hence the beginning of a BAPSIM program. Similarly, the literal occurrence of DECLARE (line 1090) signals the beginning of a structure block. The body of the structure block is made up of three declaration statements which serve to specify the machine structure. These

declaration statements are the formal counterpart of the blocks of FIGURE 1.

The first declaration statement (lines 1100 and 1110) is a register declaration statement specifying the seven registers within the object machine. The location counter, named LC, and the memory address register, named MAR, are declared to be 6-bit registers. The exchange register, named XR, the first accumulator, named AC1, and the second accumulator, named AC2, are declared to be 9-bit registers. Further, TELCOMP contains a 3-bit instruction register, named IR, and a 1-bit stop/run flip-flop, named SR. Note that commas are used to separate the individual register declarations and a semi-colon serves to end the entire register declaration statement.

Also note that the register declaration statement discussed in the preceding paragraph appears on two lines (i.e., is punched on two cards). The BAPSIM translator ignores card boundaries, column position (except when checking for a comment card), and blanks (except when checking for a literal symbol) in its scan of the input string. This allows the programmer considerable freedom in punching his machine description and permits the writing of easily readable programs.

The second statement of the structure block (line 1120) is a subregister declaration statement specifying the two

(sub) fields making up the exchange register. The left-most field, named OP, is three bits long and extends from bit zero of the exchange register through bit two. The right-most field, named ADDR, is six bits long and extends from bit three of register XR through bit eight. Implicit in these subregister declarations is information about the TELCOMP instruction format. When the exchange register contains a word to be interpreted as an instruction (as opposed to a data word) the bits zero through two are considered to be the operation code and the bits three through eight are considered to be the operand address. Thus, the TELCOMP computer is a single address machine with a maximum of eight different machine instructions.

Similarly, a data word format could be defined. The subregister declaration statement corresponding to TELCOMP's 2's complement integer data word might appear as:

```
SUBREGISTER      XR(SGN)=XR(0),XR(MAG)=XR(1-8);
```

The third statement of the structure block (line 1130) is a memory declaration statement. TELCOMP has a single 64-word, 9-bit memory, named M, which is accessed via a memory address register named MAR.

The literal occurrence of END DECLARE (line 1140) signals the end of the structure block. A note on syntax: the final statement in a block must not be followed by a semi-

colon. The delimiter indicating the end of the block also serves to indicate the end of the last statement within the block.

The structure block just described specifies the machine hardware elements of interest at the register-transfer level. The implementation details of this hardware are suppressed since they are not of interest at this level of description. In contrast to the block diagram, note that the formal description of TELCOMP has not defined the data paths between registers. However, these data paths are to be found implicitly in the transfer statements which describe the behavioral characteristics of the object machine.

The literal occurrence of `FETCH` (line 1170) signals the beginning of a fetch block. The body of the fetch block is made up of the following five transfer statements which define the fetch cycle of the machine. The first of these transfer statements (line 1180) is an example of a direct transfer between registers; the contents of the location counter `LC` are transferred unaltered to the memory address register. Implicit in this statement is the fact that a data path of some sort exists between these two registers.

The second statement in the body of the fetch block (line 1190) is an example of a memory access transfer statement. The contents of the memory word whose address is contained in the memory address register are transferred

unaltered to the exchange register. Again, a data path from the memory word to the exchange register is implied. Also note that the details of the address decoding operation are not specified or implied by this transfer statement. Such details are of interest to the logic designer or to the student of logic design, but they are concerned with a machine level "below" the register-transfer level. At the level of concern here, such details may be, and are, suppressed.

The next two statements (lines 1200 and 1210) are further examples of direct transfers. The fifth and final statement in the body of the fetch block (line 1220) is an example of a transfer statement which calls for a logical operation to be performed during the transfer operation. The logical operation is specified by the binary arithmetic operator `.ADD..` This statement may be read as "increment the location counter by one". During the addition operation the contents of the operand register LC are taken to be a binary integer and a binary integer 1 is added in the least significant (i.e., right-most) bit position. The result is stored back in register LC.

The five transfer statements making up the body of the fetch block are examples of simple statements (they do not have a label prefixed to them) and they are separated by commas. They need not have been punched on separate cards,

however, their order is important. Note that the final statement in the block is not followed by a comma. The literal occurrence of END FETCH (line 1230) signals both the end of the last statement within the block and the end of the block itself.

The literal occurrence of DECODE (line 1260) signals the beginning of a decode block. The body of the decode block is made up of eight IF-statements, separated by commas, which serve to associate with each of the eight possible integer bit patterns in the 3-bit instruction register IR the name of an object machine instruction. Each of these eight machine instruction names appears in the execute block as a label on a compound statement.

The sequence of statements making up the body of the decode block describes the TELCOMP instruction decoding operation on a functional level and suppresses all of the implementation details of interest to the logic designer. On the functional level of interest here, the instruction decoding operation merely serves to invoke the proper sequence of machine micro-operations. In this simple case, the particular sequence of micro-instructions invoked depends only upon the contents of the instruction register IR. More complex dependencies are specified by including other BAPSIM statements in the decoding sequence.

There is an implicit flow of control, or sequence of op-

erations, which is reflected in the BAPSIN program structure: the instruction fetch operation is performed first, followed by the instruction decoding operation, followed in turn by the instruction execution operation. This fetch-decode-execute (FDE) cycle is found in all present day digital computers and is reflected in the structure of the object machine simulator constructed by the BAPSIM translator.

The literal occurrence of END DECODE (line 1350) signals both the end of the last statement in the sequence making up the body of the decode block and the end of the block itself. The literal occurrence of EXECUTE (line 1380) signals the beginning of an execute block. The body of the execute block is made up of eight compound statements which serve to describe the sequence of micro-operations associated with each object machine instruction. Note that a compound statement is nothing more than a labeled sequence of simple statements. The simple statements are separated by commas; the compound statements are separated by semi-colons. The labels correspond to the instruction mnemonics; the mnemonics are associated with specific operation codes in the decode block.

The first compound statement in the execute block, labeled ADD, describes the TELCOMP addition operation. The addition operation is "executed" by TELCOMP by performing three micro-operations: an operand fetch (line 1390), a register-register add (line 1400), and a direct transfer

(line 1410).

In a similar manner, the second compound statement, labeled SUB, describes the TELCOMP subtraction operation as a sequence of seven micro-operations (line 1420 - line 1480). The micro-operation of line 1430 illustrates the use of the unary logical operator .NOT..

The TELCOMP shift right operation, SRO, is described by a sequence of three micro-operations. The third operation of this sequence (line 1510) illustrates the use of the BAPSIM functional operator .SHR.. In this operation the contents of the second accumulator, AC2, are transferred to the first accumulator, AC1, shifted one bit position to the right. The contents of the second accumulator, AC2, are left unaltered by this operation.

The TELCOMP unconditional transfer operation, TRU (line 1520), is "executed" by performing a single direct transfer micro-operation.

The TELCOMP transfer on negative accumulator operation, TRN, is specified by a single IF-statement micro-operation (line 1530). The condition for the single direct transfer is that the left-most bit of the first accumulator, AC1(0), is a '1'. If this condition is not met the direct transfer is not performed.

The next two TELCOMP machine instructions, store accumulator STA and clear accumulator CLA, are obvious. The

final instruction, however, deserves an explanation.

The purpose of the stop instruction (line 1570) is to halt the fetch-decode-execute cycle. In all cases, this operation is specified in a BAPSIM machine description by the simple statement GO TO STOP. During the execution of the simulated fetch-decode-execute cycle, one and only one machine instruction (i.e., series of micro-operations) will be invoked. Hence following the execution of a machine instruction there is an implied flow of control back to the fetch portion of the cycle -- control does not flow through more than one machine instruction as implied by the structure of the execute block. Thus, there is an implied GO TO FETCH statement after each of the compound statements making up the execute block. The correct flow of control is implemented in the object machine simulator constructed by the BAPSIM translator. See the general simulator flowchart in Appendix 5. Obviously, there must be some way to get out of the fetch-decode-execute cycle or the computer would "run" indefinitely. The statement GO TO STOP serves precisely this purpose.

The literal occurrence of END EXECUTE (line 1580) serves to signal the end of the execute block. Note that there is no semi-colon delimiter following the final statement of the execute block.

At this point in the BAPSIM program (line 1580) the

structural and behavioral characteristics of the object machine have been completely described. It remains for the programmer to specify information for use by the BAPSIM translator in constructing the object machine simulator.

The first of the two statements making up the input-output block is an initialization statement (line 1610). This statement informs the BAPSIM translator that when the simulator is run, input data corresponding to the named registers, subregisters, and memories will be provided by the programmer. The translator uses this information to include in the simulator provisions for accepting this data input. The input data supplied by the programmer defines the initial state of the machine for the purpose of simulation.

At the end of each fetch-decode-execute cycle the simulator provides printed output which may be thought of as a snapshot indicating the machine state. Those registers of interest to the programmer during the simulation run -- i.e., those registers the programmer uses to define the machine state -- are specified in a monitor statement (line 1630). This statement tells the translator that at the end of every fetch-decode-execute cycle of the simulation the contents of the six registers listed are to be printed out for inspection. Simulator input-output operations will be discussed further in a following section.

The literal occurrence of END SIMULATION (line 1660)

signals the end of the simulation block and hence the end of the BAPSIM machine description of the TELCOMP computer.

THE BAPSIM TRANSLATOR

Introduction

The purpose of the BAPSIM translator is to convert an object machine description written in the BAPSIM language into a simulator which is capable of "executing" programs written in the machine language of the machine being simulated. Since the simulator is a PL/1 program, from one point of view the BAPSIM translator may be thought of as a program-writing program.

Once the source language has been defined, three questions must be answered prior to the development of any translator/compiler:

1. What programming language is the translator/compiler to be written in?
2. What is the target language?
3. What translation technique is going to be used?

Answers to the first two of these questions will be briefly discussed next. The third question will be dealt with in more detail later.

In the case of compilers, the target language is usually the machine language of a particular machine (or series of machines) and, traditionally, compilers have been written in

the assembly language of the specified machine(s).

Recently, however, the use of higher level programming languages in the implementation of complex software systems has been gaining in popularity, as the advantages of such a method have come to be known and accepted. Certainly, not the least of these advantages has to do with the time required to write and debug large programs. It is simply easier to design and implement a large program, or group of programs, in a higher level programming language than it is to implement the same system in assembly language. As an example of this approach, one could cite the MULTICS supervisory system which was written in PL/1. In addition, there exists at least one FORTRAN compiler which was written in PL/1.

In order to explore the possibilities of such a technique, the decision was made to write the translator/compiler for the BAPSIM simulation system in a high level language. And, since the process of translation/compilation is, for the most part, a symbol manipulation process, PL/1 was selected as the language to be used. This decision was based primarily on PL/1's extensive and exceptional facilities for character processing.

The second question to be answered was: what is the target language? Unconventionally, but very successfully, PL/1 was also chosen as the target language. There were two reasons for this choice: (1) the nature of the simulation

process being implemented, and (2) the desire for program mobility.

The BAPSIM simulation system is intended to model digital machines at the register-transfer level. At this level, the fundamental data item is the bit string; one of PL/1's basic data items is the bit string.

One of the reasons for using higher level programming languages is the ability to run programs in the language on different machines -- i.e., program mobility. Such mobility is essentially lost in compilers or translators if the target language is not equally mobile. A FORTRAN compiler written in PL/1 is not really mobile if it generates S/360 machine code. Of course, mobility is not usually required in a compiler. However, in a translator or monitor it can be most useful.

In summary, by choosing PL/1 as both the language of the translator and as the target language, complete mobility of the simulation system is assured.

The question of translation technique remains to be answered. The first section below will deal only with the syntactic analysis phase of the translator. The equally important semantic analysis portion of the translator will be dealt with in a following section.

Syntactic Analysis Techniques

Syntactic analysis techniques fall into two broad categories: (1) general algorithms, and (2) special algorithms.

General algorithms are those which allow the syntactic analysis of a whole class of computer languages for which the syntax can be expressed with the aid of some code or formalism. These algorithms require two kinds of data: (1) the syntactic rules of the particular source language being analyzed, and (2) the source program to be analyzed.

Special algorithms (for syntactic analysis) are those with a structure which reflects the syntax of a particular language and which are therefore applicable only to that language. These algorithms require only one kind of input data - the source program to be analyzed.

Since the BAPSIM language discussed above is the only language of interest in the work reported on here, generality in the translator was not required. Further, the efficiency (i.e., fast parsing speed) and ease of use associated with translators implementing one of the special algorithms were considered to be desirable.

Syntactic analysis techniques may also be classified according to the particular parsing algorithm they employ. Top-to-bottom parsers begin with the most general syntactic unit in the language grammar and, by making successive

substitutions for the non-terminal syntactic units, attempt to produce (construct) the given input string. In opposition, bottom-to-top parsing algorithms start with the given input string and, by applying the productions of the language grammar backwards, attempt to recognize the input as an occurrence of the most general syntactic unit.

General comparisons between the two parsing methods are difficult for two reasons. First, parsing efficiency is more a function of the language to be parsed than of the parsing method used. Some languages may be parsed more efficiently using the top-to-bottom method and others may be parsed more efficiently using the bottom-to-top method. Simple languages may be parsed efficiently by either method.

Second, the top-to-bottom method does have one drawback. It does not work for language definitions that are left recursive. This defect is not too important, however, since it is always possible to transform a given left recursive language definition into an equivalent grammar which is not left recursive (26).

Since the BAPSIM language definition is not left recursive, the choice of parsing method to be employed in the translator was dictated by another consideration discussed in the next section.

The Method of Syntactic Functions

The special algorithm which was selected for use in the BAPSIM translator is referred to in the literature as the method of syntactic functions. This top-to-bottom approach to the syntax analysis problem associates with each syntactic definition in the language a corresponding Boolean function in the translator. This direct relationship between the syntax of the language and the structure of the translator makes the translator easy to write initially and easy to modify if the syntax of the language is altered or extended.

To illustrate the method of syntactic functions, consider the problem of analyzing (in this case recognizing) lists of identifiers. In Backus-Normal form, the syntactic definition of a list might appear as follows:

$$\langle \text{list} \rangle ::= \langle \text{identifier} \rangle \mid ', ' \langle \text{list} \rangle$$

$\langle \text{list} \rangle$ and $\langle \text{identifier} \rangle$ are syntactic units and the notation $', '$ is used to indicate the literal occurrence of a comma. In words, this definition states that "the syntactic unit $\langle \text{list} \rangle$ is defined as any occurrence of the syntactic unit $\langle \text{identifier} \rangle$ or a literal occurrence of a comma followed by any occurrence of the syntactic unit $\langle \text{list} \rangle$ ". Note that in Backus-Normal form, $\langle \text{list} \rangle$ is defined in terms of itself. However, the definition is not ambiguous being "grounded" by

the first alternative in the above definition.

The syntactic function (procedure) corresponding to this syntactic definition is given in PL/1 as:

```

LIST: PROCEDURE RETURNS(BIT(1)) RECURSIVE;
      IF IDENTIFIER THEN
        IF COMMA THEN
          IF LIST THEN GO TO TRUE;
            ELSE GO TO FALSE;
          ELSE GO TO TRUE;
            ELSE GO TO FALSE;
        FALSE: RETURN('0'B);
          GO TO EXIT;
        TRUE: RETURN('1'B);
      EXIT: END LIST;

```

where IDENTIFIER is another Boolean procedure which returns true ('1'B) if the character string pointed to by the text pointer is an identifier, and returns false ('0'B) otherwise. When an identifier is found, and before true is returned to the calling procedure, the text pointer is advanced one position beyond the identifier. COMMA is another Boolean procedure which returns true if the character pointed to by the text pointer is a ',' and returns false otherwise. Again, the text pointer is advanced one position if and only

if a comma is found.

For those readers familiar with PL/1 or ALGOL it should be evident how the structure of this procedure reflects the structure of the syntactic unit being analyzed (<list>).

To illustrate how the choice of meta-language influences the syntax analysis process, consider the same example using the meta-language developed above. Using this meta-notation, the syntactic definition of a list would be as follows:

```
LIST = IDENTIFIER $ ( ',' IDENTIFIER )
```

In words this definition states: "a list is defined to be an identifier followed by zero or more occurrences of the combination comma-identifier".

The most important thing to note about this definition is that the definition of a list is no longer recursive -- i.e., LIST is no longer defined in terms of itself.

The syntactic function (procedure) corresponding to this second syntactic definition is given in PL/1 as:

```
LIST: PROCEDURE RETURNS(BIT(1));
      IF -IDENTIFIER THEN GO TO FALSE;
      L1: IF -COMMA THEN GO TO TRUE;
         IF -IDENTIFIER THEN GO TO FALSE;
         GO TO L1;
FALSE: RETURN('0'B);
```

```
GO TO EXIT;  
TRUE: RETURN('1'B);  
EXIT: END LIST;
```

where IDENTIFIER and COMMA are as defined in the earlier example. Note that since the definition of LIST is no longer recursive, the corresponding syntactic function (procedure) need not be recursive.

Again, note how the structure of the syntactic unit (LIST) under consideration is reflected in the structure of the procedure. In the second example, an iterative technique has been used instead of the recursive technique of the first example. The change from recursion to iteration is not always possible however. In the cases requiring recursion -- for example, the CONDITION portion of the BAPSIM IF-statement -- the RECURSIVE option allowed with PL/1 procedures considerably simplifies the problem of implementation.

Implementation Details

With this general discussion of method out of the way we may proceed to consider the implementation details of such a technique. Appendix 3 contains a program listing of the syntax analysis portion of the BAPSIM translator and should be consulted while reading this section.

With reference to Appendix 3, note that the BAPSIM syntax analyzer contains three functionally distinct sections of code: (1) code for the necessary housekeeping chores (lines 1080 - 1960), (2) the main part or heart of the program (lines 2000 - 2100 and lines 7960 - 7980), and (3) the Boolean function procedures. These procedures fall into two categories: (1) primitive procedures, which recognize the primitive elements of the BAPSIM vocabulary and handle error conditions, and (2) the non-primitive procedures which correspond one-for-one with the syntactic units of the BAPSIM language.

One of the most important housekeeping chores involves reading the BAPSIM machine description. The input data is considered to be a finite character string which, of necessity, is punched onto a sequence of cards. The cards are read one at a time into an 80-character buffer, named CARD, and made available from there to the rest of the program. Also associated with the task of reading the input stream is a text pointer, named I, which keeps track of the correct position in the text being read and analyzed. As the input string is read, the text pointer is advanced by the various primitive procedures and when the end of the buffer is reached an end-of-card (EOC) condition is raised.

This condition causes four things to occur. First, the buffer CARD is refilled with the next eighty characters of

the input string. Second, the first character of the card just read is examined. If the character is an asterisk '*', indicating a comment card has been read, the comment is printed in the source listing and another card is read into the buffer. If the character is not an asterisk, the statement counter is incremented by one and the statement number and card are printed in the source listing. Third, the text pointer is reset to the beginning of the buffer. Finally, control transfers back to the primitive procedure which was responsible for raising the end-of-card condition and the syntax analysis process continues.

The heart of the BAPSIM translator is the single PL/1 IF-statement on lines 2060 and 2070. This statement serves to invoke all of the necessary procedures, in the proper order, for determining whether or not a valid BAPSIM program has been supplied as input. The elegant simplicity of this syntactic analysis technique should now be apparent.

Primitive procedures

The first of the primitive procedures is a one-parameter Boolean function labeled LITERAL. This procedure compares the character string passed to it by a calling procedure with the character string (of the same length) appearing at the current position of the text pointer. Note that extraneous blanks are ignored before the comparison is made. Also note

that the literal being tested for may not extend across a card boundary. If a literal occurrence of the character string passed to it is found at the current position of the text pointer, the function LITERAL returns true ('1'B) to the calling procedure after advancing the text pointer beyond the string just recognized. If no such match is found, the text pointer is not advanced and LITERAL returns false ('0'B) to the calling procedure.

In the examples above illustrating the method of syntactic functions, the existence of a Boolean function named COMMA was assumed. Implementing such a function could be accomplished by calling LITERAL with a comma as the argument -- i.e., LITERAL(','). Thus, LITERAL is a general Boolean function for recognizing the literal occurrence of any character string passed to it.

The second of the primitive procedures is a zero-parameter Boolean function labeled ID. This procedure checks for the occurrence, at the current position of the text pointer, of a valid BAPSIM identifier. Again, extraneous blanks preceding the identifier are ignored. When a valid BAPSIM identifier is recognized it is stored in a varying length character string named SYMBOL. If the identifier is greater than eight characters in length, a new character string is stored in SYMBOL which consists of the first four characters of the original identifier concatenated with the

last four characters of the original identifier. Thus, in practice, BAPSIM identifiers are limited to character strings of length eight. During the syntactic analysis phase of the translation process, the identifier stored in SYMBOL by the procedure ID is never used. It is used, however, during the semantic evaluation phase of the translation process.

The third of the primitive procedures is a zero-parameter Boolean function labeled INT. This procedure checks for the occurrence of a valid BAPSIM integer. Its operation is analogous to the operation of procedure ID except that integers are not "shrunk" if they exceed eight characters. Again, the integer stored in SYMBOL is never used in the syntactic analysis phase of the translation process.

The fourth of the primitive procedures is a zero-parameter procedure labeled ERROR. This procedure is called whenever the character string at the current position of the text pointer does not correspond to the character string expected by the syntax analyzer -- i.e., whenever an invalid machine description is provided as input to the analyzer. Error handling is extremely simple. The offending card column is flagged with a dollar sign '\$' and the translation process is aborted. For a language as simple syntactically as BAPSIM this technique has proven itself to be adequate.

Three additions to the error handling routine were considered and all were rejected as being unnecessary for the

purposes of the work reported on here. The first addition would have called the error routine with an integer argument whenever an input syntax error was detected. The error routine would then output an error message, determined by the integer passed to it, which contained more complete diagnostic information. The second addition would have been to provide some sort of error recovery procedure. The complexity of such a procedure to handle all possible errors was considered to be prohibitive to its implementation. Another possibility would have been simply to advance the text pointer beyond the offending statement (after flagging the error) and continuing with the translation process. This provision would have made it possible to detect more than one error on each run of the translator. Further development of the BAPSIM translator will certainly include this provision.

Non-primitive procedures

The remaining section of the syntax analyzer consists of the fifty-four function procedures corresponding to the fifty-four non-primitive syntactic units of the BAPSIM language. By comparing any syntactic procedure with its associated syntactic definition the relationship between the structure of the definition and the structure of the procedure should be apparent. It is precisely this direct correspondence which results in a syntax analyzer that is easily modified to

reflect changes in, and extensions to, the language being analyzed.

Two of the fifty-four non-primitive procedures (TERMINAL_STMT, TRANSFER_STMT) are extraordinary. The reason for the added complexity of these two procedures is that the BAPSIM language is not defined so that each choice among alternative syntactic units can be made by examining only the first element of that syntactic unit. In other words, certain character strings in the language may represent the occurrence of more than one syntactic unit of the language. Precisely which syntactic unit a given instance of a character string represents depends on the context in which the character string appears.

More specifically, character strings of the form .ID or .ID(.INT) may represent the occurrence of either the syntactic unit DEST or the syntactic unit TERMINAL. If a character string having one of these two forms is followed by the sequential operator '<-' it is taken to be an occurrence of the syntactic unit DEST. However, if the character string is followed by the sequential operator ':=' it is taken to be an occurrence of the syntactic unit TERMINAL.

Consider the problem of recognizing a terminal statement. The BAPSIM translator first checks for a transfer statement (since transfer statements occur more frequently in BAPSIM machine descriptions than terminal statements). If we

have a valid terminal statement, the translator recognizes the initial portion of the statement as an occurrence of the syntactic unit DEST and then checks for the sequential operator '<'. Of course, the transfer operator is not found (remember we have assumed that we have a valid terminal statement) and so the procedure TRANSFER_STMT must report back to its calling procedure (SIMPLE_STMT) a failure. At this point, however, the text pointer would be pointing just beyond the character string incorrectly identified as an occurrence of DEST. If the translator were now merely to check for a terminal statement, failure would result since the translator is looking for an occurrence of TERMINAL and the text pointer is pointing to the sequential operator ':='.

The solution to this problem is to reset the text pointer to the beginning of the terminal statement before TRANSFER_STMT reports back a failure. With this provision included, the translator continues by checking for and finding an occurrence of TERMINAL, followed by an occurrence of ':=', followed by the remainder of the terminal statement.

The Simulator

The simulator constructed by the BAPSIM translator is a PL/1 program which is capable of "executing" programs written in the machine language of the object machine. The input to

the simulator consists of bit strings specifying the initial state of the object machine's hardware elements. The output of the simulator is a series of snapshots of the object machine state at the end of each fetch-decode-execute cycle.

The purpose of the semantic evaluation portion of the BAPSIM translator is to generate the PL/1 simulator. As may be seen by referring to the program listing of the complete BAPSIM translator in Appendix 4, the semantic evaluation code is interspersed with the syntactic analysis code. The reason for this is that the code for the PL/1 simulator is generated "on the fly" as the syntactic analysis process is carried out. With this technique, elements of the PL/1 simulator are generated as soon as their corresponding elements in the machine description are recognized. In what follows, the more important segments of the semantic evaluation code will be discussed.

Primitive procedures

Two additional primitive procedures are associated with generating the PL/1 simulator. The first of these is a zero-parameter procedure labeled STAR. Recall that whenever an identifier or integer is recognized it is stored in a varying length character string called SYMBOL. The purpose of the procedure STAR is simply to retrieve the last identifier or integer recognized -- i.e., to retrieve the current contents

of SYMBOL.

The second of the primitive semantic evaluation procedures is a one-parameter procedure labeled OUT. This procedure is responsible for actually "writing" the PL/1 simulator. As individual statements of the simulator are generated, they are passed to OUT. This procedure converts them to 80-character records and writes them out into a file named ASSM. This file is subsequently read back in, as input to the PL/1 compiler, when the simulator is compiled prior to execution. In addition, procedure OUT prints out the code being generated for the simulator alongside the BAPSIM source code. This juxtaposition is useful in illustrating the relationships between elements of the source program and corresponding elements of the PL/1 simulator. Procedure OUT also keeps track of the number of semantic action records generated during the translation process. The final count is printed out as an item of statistical information at the conclusion of the translation process.

Code generation

Referring to the translator program listing in Appendix 4, observe that the occurrence of SIMULATION (line 1426) in the BAPSIM machine description gives rise to the first two lines of simulator code. Similarly, when the translator recognizes END SIMULATION (line 1440) the final five lines of

simulator code are generated.

The code generated by the translator upon recognition of END DECLARE is a bit unusual. See line 1482 in Appendix 4. The PL/1 %INCLUDE statement is a preprocessor statement which is executed during a preliminary scan of the PL/1 simulator. The statement is used here to incorporate strings of external text into the source program (i.e., the PL/1 simulator) being scanned. The external text to be included must be a member of a partitioned data set. The identifier in the %INCLUDE statement (in this case DCLS) specifies the name of the data set member which is to be included at this point in the PL/1 source text.

The data set member DCLS contains the PL/1 code for the declarations which are common to every simulator constructed by the translator. Of course, this declaration code could have been generated directly by the translator. The indirect method was selected because it decreased the size of the BAPSIM translator.

Two other %INCLUDE statements are generated by the BAPSIM translator. The occurrence of END DECODE causes %INCLUDE ENDCODE to be created (line 1554). The data set member ENDCODE contains the PL/1 code required for the simulator to handle the occurrence of illegal operation codes during the instruction decoding operation. See Appendix 7. This external text is incorporated into every simulator con-

structed by the translator.

The occurrence of END EXECUTE causes %INCLUDE PRIMPROC to be generated (line 1618). The data set member PRIMPROC contains the PL/1 code required to implement fourteen of the primitive object machine operations. See Appendix 8. The text for all of these operations is included in every simulator constructed by the translator even though the simulator for a particular object machine may not require some of them. Admittedly, keeping track of the primitive operations required by a particular simulator and then including only those operations would be a more elegant approach to the problem. However, it would also be a more complex and costly approach than the one chosen here.

Simulator_I/O

The translator procedure INITIALIZATION_STMT (line 1640) is responsible for creating the PL/1 code which handles the input operations for the simulator. The code generated here determines what form the simulator input initialization data must have. The first datum, required by every simulator, is associated with the GET LIST(CH#CK) statement (line 1654). The integer expected by this statement specifies the maximum number of object machine fetch-decode-execute cycles the simulator is to execute.

Each of the identifiers in a BAPSIM initialization list,

corresponding to those object machine hardware elements whose initial state is to be specified, gives rise to a GET DATA statement in the simulator. Hence, the format for specifying the initial state of the object machine is a sequence of PL/1 data lists. The PL/1 reference manual gives a complete description of the rules for supplying data to a GET DATA statement, obviating the necessity of going into the details here.

The object machine machine-language program is input by specifying an initial memory state. That is, the simulator reads the machine-language program into (simulated) memory and then executes it.

The translator procedure MONITOR_STMT (line 1694) is responsible for creating the PL/1 code which handles the output operations for the simulator. At the end of each simulated fetch-decode-execute cycle a snapshot of the object machine state is printed out. The first item printed out is the cycle number. Following the cycle number appears a printed listing of the bit patterns contained in each of the object machine hardware elements called out in the BAPSIM monitor list.

If timing information has been specified in the BAPSIM machine description, the time at the end of each fetch-decode-execute cycle will be printed out immediately following the cycle number. The time, recall, is in relative units.

Primitive operations

The simulator implements fourteen of the sixteen primitive BAPSIM machine operations by means of procedure calls. The two operations direct transfer and logical complement are implemented directly since PL/1 includes them as primitive operations. The remaining fourteen primitive operations must be implemented indirectly since they have no counterparts in the PL/1 language. It is this latter fact which serves as a justification for the development of a new programming language. To the BAPSIM user the primitive language operations and the primitive object machine operations are the same. The user is thus able to describe a given object machine in a straightforward, natural manner. It is a case of providing the user with a notational system which matches exactly the problem to be described.

The details of implementing the basic object machine operations are of no concern to the BAPSIM user since they are transparent to him. They are included here for the sake of completeness and with the hope that they may be of interest to the student of language implementation. Appendix 8 is a listing of the partitioned data set member PRIMPROC which contains the code for the fourteen primitive machine operation procedures.

The procedure #SET is invoked whenever a register,

subregister, or memory word is to be set to a specific bit pattern. Clearing a register or setting a flip-flop (i.e., a single-bit register) are its most common uses. The BAPSIM statement `A <- 0` gives rise to the statement `CALL #SET(A,0);` in the simulator. When this latter statement is executed, during the simulation of the object machine, the procedure `#SET` will clear register A. Similarly, the BAPSIM statement `C <- 7` gives rise to the statement `CALL #SET(C,7);`. When this statement is executed, the integer 7 is converted to its binary equivalent (`'111'B`) and this binary value is stored in register C. The assignment of the binary value to register C is from the right. That is, if register C is five bits long, the value stored in C as a result of this statement is `'00111'B`. If register C is two bits long, the value stored in C is `'11'B`.

The procedure `UA#D` is invoked whenever a unary-and operation is specified. The BAPSIM statement `C <- .AND. X` gives rise to the statement `CALL UA#D(C,X);` in the simulator. When this latter statement is executed, the one-bit register C will be set to `'1'` if all of the bits of register X are `'1'`. Register C will be set to `'0'` otherwise. If register C is a multi-bit register a length error will result during the simulation run since the unary-and operation is undefined in such a case.

The procedure `#UOR` is invoked whenever a unary-or opera-

tion is specified. The statement `D <- .OR. Y` gives rise to the statement `CALL #UOR(D,Y)`; in the simulator. When this latter statement is executed, the one-bit register D will be set to '1' if any of the bits of register Y are '1'. Register D will be set to '0' otherwise. If register D is a multi-bit register a length error will result during the simulation run since the unary-or operation is undefined in such a case.

The procedure US#L is invoked whenever a shift left operation is specified. The statement `A <- .SHL. Q` gives rise to the statement `CALL US#L(A,Q)`; in the simulator. When this latter statement is executed, the contents of register Q are transferred to register A shifted one bit to the left. The right-most bit of register A is filled with a '0'. The contents of register Q are unaltered by this operation. If the two registers involved in this transfer operation are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure US#R is invoked whenever a shift right operation is specified. The statement `A <- .SHR. Q` gives rise to the statement `CALL US#R(A,Q)`; in the simulator. When this latter statement is executed, the contents of register Q are transferred to register A shifted one bit position to the right. The left-most bit of register A is filled with a '0'. The contents of register Q are unaltered by this operation.

If the two registers involved in this transfer operation are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure CI#L is invoked whenever a circulate left operation is specified. The statement `A <- .CIRL. Q` gives rise to the statement `CALL CI#L(A,Q);` in the simulator. When this latter statement is executed, the contents of register Q are transferred to register A shifted end-around one bit position to the left. The contents of register Q are unaltered by this operation. If the two registers involved in this transfer operation are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure CI#R is invoked whenever a circulate right operation is specified. This procedure is analogous to the procedure which implements the circulate left operation.

The procedure #ADD is invoked whenever a binary addition operation is specified. The statement `A <- A .ADD. X` gives rise to the statement `CALL #ADD(A,A,X);` in the simulator. When this latter statement is executed, the contents of register X are added, in a strict binary sense, to the contents of register A and the result is stored back in register A. If the result of the addition operation is larger than the maximum binary integer register A is capable of holding, a

special one-bit flag, named OVERFLOW, is set to '1'. This overflow indicator is available to the BAPSIM programmer without an explicit declaration. No other BAPSIM program element may be assigned this name. Hence, OVERFLOW is a reserved word in the BAPSIM language from the user's viewpoint.

The second operand associated with the addition operator may also be an integer. The statement `L <- L .ADD. 1` gives rise to the statement `CALL #ADD(L,L,1)`; in the simulator. When this latter statement is executed, the contents of register L (treated as a binary integer) are incremented by one. OVERFLOW is set to '1' if the addition operation results in an overflow out of the left-most bit position of register L.

The procedure #SUB is invoked whenever a binary subtraction operation is specified. The statement `C <- C .SUB. 1` gives rise to the statement `CALL #SUB(C,C,1)`; in the simulator. When this latter statement is executed, the contents of register C (treated as a binary integer) are decremented by one. If the result of the subtraction operation is less than zero an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure #BOR is invoked whenever a binary-or operation is specified. The statement `A <- A .OR. X` gives rise to the statement `CALL #BOR(A,A,X)`; in the simulator. When this latter statement is executed, the contents of register X

are or-ed bit-by-bit with the contents of register A and the result is stored back in register A. If the two register operands are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure BA#D is invoked whenever a binary-and operation is specified. The statement `A <- A .AND. X` gives rise to the statement `CALL BA#D(A,A,X)`; in the simulator. When this latter statement is executed, the contents of register X are and-ed bit-by-bit with the contents of register A and the result is stored back in register A. If the two register operands are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure BX#R is invoked whenever an exclusive-or operation is specified. The statement `A <- A .XOR. Y` gives rise to the statement `CALL BX#R(A,A,X)`; in the simulator. When this latter statement is executed, the contents of register X are exclusive-or-ed bit-by-bit with the contents of register A and the result is stored back in register A. If the two register operands are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure BS#L is invoked whenever a multiple shift left operation is specified. The statement `A <- X .SHL. 4`

gives rise to the statement `CALL BS#L(A,X,4)`; in the simulator. When this latter statement is executed, the contents of register X are transferred to register A shifted four bit positions to the left. The contents of register X are unaltered by this operation. If the two registers involved in this transfer operation are not the same length an error message will be printed during the simulation run indicating an invalid operation has been performed.

The procedure `BS#R` is invoked whenever a multiple shift right operation is specified. This procedure is analogous to the procedure which implements the multiple shift left operation.

CONCLUSIONS

Work on the BAPSIM programming system has demonstrated the possibility of developing a means for specifying and simulating the behavior of digital machines. In addition, the successful application of the techniques used in implementing the simulation system suggests that the case for special purpose programming languages and their associated language processors may be stronger than previously thought.

The main advantage of special purpose languages, that a language tailored to a certain class of problems is easier to learn and use than a general purpose programming language, has been recognized for some time. The main disadvantage, that a special purpose language processor must be provided along with every special purpose language if it is to have any practical application, has been known for an equally long time. And it is the time and effort required to provide the language processor that have been deciding negative factors in most decisions regarding the development of special purpose programming systems.

The work done in developing the BAPSIM simulation system has shown that the development of special purpose language processors need not be the expensive, time-consuming, often ad hoc procedure it has been in the past. Use of the meta-

language employed in the definition of the BAPSIM language together with the method of syntactic functions results in a straightforward, clearly defined technique for developing the language processor. In fact, if the language processor is written in a high level programming language with symbol manipulation capabilities, development of the processor will probably no longer be the most time consuming part of the development of a special purpose programming system. Using the BAPSIM simulation system as an example, it is the language definition phase of the development procedure which will require the most attention and time. The reason for this, of course, is that there now exists no algorithmic procedure for use in the definition of a language. And until there exists such a procedure, languages will continue to be developed in a trial and error, iterate and correct manner.

The BAPSIM simulation language was developed in just such a way. The work of others in the field of computer description languages was examined and their results used as a starting point. Features common to all previous efforts were extracted and formalized into a machine-readable notation. Additional language elements were included as dictated by the fact that the end result was to be a computer description language to be used in the simulation of machine behavior. An attempt was then made to describe a simple machine (TELCOMP) in the language. Language deficiencies were noted

and appropriate additions and corrections were made. The updated version of the language was then applied to a more complex machine (Bartee, Lebow, and Reed, Chapter 9). Inadequacies in the language were again observed and corrective measures were taken. This trial and error, design and analyze technique was continued through two more machines, the Fabri-Tek Bi-Tran Six and the Digital Equipment Corporation PDP-8. The BAPSIM language and translator have been tested on all four of the above mentioned machines. Numerous machine language programs were run and checked for each of these machines in the process of debugging the translator and the simulators which it constructs. The simulation system has proven itself capable of describing and simulating general purpose digital processors of sufficient complexity to justify its development and to attract the attention of students of computer design.

The BAPSIM simulation system has also demonstrated the advantages to be gained by the use of higher level programming languages in the development of special purpose programming systems. Systems programs -- compilers, translators, monitors, and the like -- need no longer be written in assembly language. Current high level languages, such as PL/1, provide the systems programmer with virtually all of the manipulative power available to the user of assembly languages and in a form which is easier to utilize. The result

is a decrease in the time required to write and debug the systems program involved.

In the case of special purpose language translators, generating "object" code in a higher level language also has demonstrated merit. The most obvious advantage such a technique has to offer is that a processor for the generated code already exists and need not be written. The associated disadvantage, of course, is that an existing general purpose processor for the generated code is probably not as efficient as a special purpose processor written specifically for the generated code. For translators which do not generate large quantities of "object" code, such as the BAPSIM translator, use of an existing compiler is to be preferred.

Any answer to the question of efficiency in the processing of the generated "object" code must also take into account the availability of extremely fast compilers for existing high level programming languages. All (except the best systems programmers) who have at their command the use of a high level compiler such as the WATFOR or WATFIV FORTRAN compilers or the Cornell PL/1 compiler would be ill-advised to overlook the possibility of including one of these general purpose language processors in their special purpose programming systems. The best systems programmers, of course, will write special purpose language processors which are capable

of directly generating executable code, thereby eliminating the need for the second processor.

BIBLIOGRAPHY

1. Bartee, T., Lebow, I. and Reed, I. S. Theory and Design of Digital Machines. McGraw-Hill, New York. 1962.
2. Bashkow, J. R., Friets, J. and Karson, A. A programming system for detection and diagnosis of machine malfunctions. IEEE Trans. Electronic Computers EC-12: 10-17. 1963.
3. Bell, C. G. and Newell, A. Computer Structures: Readings and Examples. McGraw-Hill, New York. 1971.
4. _____ . The PMS and ISP descriptive systems for computer structures. Joint Computer Conference Proceedings, Spring 36: 351-374. 1970.
5. Bobrow, D. G. Symbol Manipulation Languages and Techniques. North Holland, Amsterdam. 1968.
6. Bolliet, L. Compiler writing techniques. In Programming Languages, Genuys, F., Ed. 113-289. Academic Press, London. 1968.
7. Breuer, M. A. Techniques for the simulation of computer logic. Communications of the ACM 7: 443-446. 1964.
8. Brooker, R. A. and Morris, D. A general translation program for phrase structure languages. Journal of the ACM 9: 1-10. 1962.
9. Buxton, J. N. Simulation Programming Languages. North Holland, Amsterdam. 1968.

10. Cheatham, T. E. and Satley, K. Syntax directed compiling. Joint Computer Conference Proceedings, Spring 25: 31-57. 1964.
11. Chu, Y. An ALGOL-like computer design language. Communications of the ACM 10: 607-615. 1965.
12. Clancy, J. J. and Fineberg, M. S. Digital simulation languages: A critique and guide. Joint Computer Conference Proceedings, Fall 27: 23-36. 1965.
13. Elgot, C. C. and Robinson, A. Random access stored program machines, an approach to programming languages. Journal of the ACM 11: 365-399. 1964.
14. Fabri-Tek, Inc. Bi-Tran Six technical and operation manual. Fabri-Tek, Inc., Minneapolis, Minn. 1967.
15. Falkoff, A. D., Iverson, K. E. and Sussenguth, E. H. A formal description of SYSTEM/360. IBM Systems Journal 3: 198-263. 1964.
16. Feldman, J. and Gries, D. Translator writing systems. Communications of the ACM 11: 77-113. 1968.
17. Flores, I. Computer Organization. Prentice-Hall, Englewood Cliffs, N. J. 1969.
18. Gavrilov, M. A. and Zakrevskii, A. D. LYAPAS: A Programming Language for Logic and Coding Algorithms. Academic Press, New York. 1969.

19. Gorman, D. and Anderson, J. P. A logic design translator. Joint Computer Conference Proceedings, Fall 22: 251-261. 1962.
20. Hellerman, H. Digital Computer System Principles. McGraw-Hill, New York. 1967.
21. IBM Corporation. IBM System/360 PL/1 Reference Manual. IBM Corporation, White Plains, N. Y. 1968.
22. Iverson, K. E. A Programming Language. Wiley, New York. 1962.
23. Katz, J. H. Optimizing bit-time computer simulation. Communications of the ACM 6: 679-685. 1963.
24. Knuth, D. E. and Bigelow, R. H. Programming languages for automata. Journal of the ACM 14: 615-635. 1967.
25. Knuth, D. E. and McNeley, J. L. SOL -- A symbolic language for general purpose systems simulation. IEEE Trans. Electronic Computers EC-13: 401-414. 1964.
26. Kurki-suonio, R. On top-to-bottom recognition and left recursion. Communications of the ACM 9: 527-528. 1966.
27. Naur, P. Revised report on the algorithmic language ALGOL 60. Communications of the ACM 6: 1-17. 1963.

28. Parnes, D. L. A language for describing the functions of a synchronous system. Communications of the ACM 9: 72-96. 1966.
29. Proctor, R. M. A logic design translator experiment demonstrating relationships of language to systems and logic design. IEEE Trans. Electronic Computers EC-13: 422-430. 1964.
30. Raphael, B. The structure of programming languages. Communications of the ACM 9: 67-71. 1966.
31. Schlaeppli, H. A formal language for describing machine logic timing and sequencing (LOTIS). IEEE Trans. Electronic Computers EC-13: 439-448. 1964.
32. Schorr, H. Computer-aided digital system design and analysis using a register transfer language. IEEE Trans. Electronic Computers EC-13: 730-737. 1964.
33. Stabler, E. P. System description languages. IEEE Trans. Computers C-19: 1160-1173. 1970.
34. Stockwell, G. N. Computer logic testing by simulation. IRE Trans. Military Electronics MIL-6: 275-282. 1962.

APPENDICES

Appendix 1. The BAPSIM Syntax


```

COMPOUND_STMT = .ID ':' SIMPLE_STMT $ ( ',' SIMPLE_STMT ) APPA1390
SIMPLE_STMT = ( DO_STMT | IF_STMT | GO_TO_STMT | TRANSFER_STMT APPA1400
                TERMINAL_STMT ) ( '<' .INT '>' | .EMPTY ) APPA1410
DO_STMT = 'DO' .ID APPA1420
IF_STMT = 'IF' CONDITION 'THEN' SIMPLE_STMT APPA1430
CONDITION = '(' BOOLEAN_TERM ')' APPA1440
BOOLEAN_TERM = BOOLEAN_FACTOR $ ( '.OR.' BOOLEAN_FACTOR ) APPA1450
BOOLEAN_FACTOR = BOOLEAN_SEC $ ( '.AND.' BOOLEAN_SEC ) APPA1460
BOOLEAN_SEC = ( '.NOT.' | .EMPTY ) BOOLEAN_PRIM APPA1470
BOOLEAN_PRIM = LOGICAL_VALUE | RELATION | '(' BOOLEAN_TERM ')' APPA1480
RELATION = SAE ( REL_OP SAE | .EMPTY ) APPA1490
REL_OP = '.EQ.' | '.NE.' | '.LT.' | '.GT.' | '.LE.' | '.GE.' APPA1500
SAE = .INT | DEST APPA1510
LOGICAL_VALUE = '.TRUE.' | '.FALSE.' APPA1520
GO_TO_STMT = 'GO TO' .ID APPA1530
TRANSFER_STMT = DEST '<-' ( .INT | UNARY_EXP | BINARY_EXP ) APPA1540
DEST = .ID( '(' (.ID ')'|.INT(')'|'-' .INT ')')| .EMPTY ) APPA1550
UNARY_EXP = UNARY_OP UNARY_SOURCE APPA1560
UNARY_OP = '.AND.' | '.OR.' | '.NOT.' | '.SHL.' | '.SHR.' | APPA1570
            '.CIRL.' | '.CIRR.' APPA1580
UNARY_SOURCE = .ID( '(' (.ID ')'|.INT(')'|'-' .INT ')')| .EMPTY ) APPA1590
BINARY_EXP = OP1 ( BINARY_OP ( OP2 | .INT ) | .EMPTY ) APPA1600
BINARY_OP = '.ADD.' | '.SUB.' | '.OR.' | '.AND.' | '.XOR.' | APPA1610
            '.SHL.' | '.SHR.' | '.CNT.' APPA1620
OP1 = .ID( '(' (.ID ')'|.INT(')'|'-' .INT ')')| .EMPTY ) APPA1630
OP2 = .ID( '(' (.ID ')'|.INT(')'|'-' .INT ')')| .EMPTY ) APPA1640
TERMINAL_STMT = TERMINAL ':=' LABEL APPA1650
TERMINAL = .ID ( '(' .INT ')' | .EMPTY ) APPA1660
LABEL = TERM $ ( BOP TERM ) APPA1670
TERM = DEF1 | DEF2 APPA1680
DEF1 = .ID ( '(' .INT ')' | .EMPTY ) APPA1690
DEF2 = '(' '-' DEF1 ')' APPA1700
BOP = '*' | '+' APPA1710

```

Appendix 2. A Sample Program


```

ADD: XR <- M(MAR),
      AC2 <- XR .ADD. AC1,
      AC1 <- AC2;
SUB: XR <- M(MAR),
      XR <- .NOT. XR,
      AC2 <- XR .ADD. AC1,
      XR <- 1,
      AC1 <- AC2,
      AC2 <- XR .ADD. AC1,
      AC1 <- AC2;
SRO: XR <- 0,
      AC2 <- XR .ADD. AC1,
      AC1 <- .SHR. AC2;
TRU: LC <- MAR;
TRN: IF ( AC1(0) .EQ. 1 ) THEN LC <- MAR;
STA: XR <- AC1,
      M(MAR) <- XR;
CLA: AC1 <- 0;
STP: SR <- 0, GO TO STOP
END EXECUTE
*
* INITIALIZE          LC,M,IR,MAR,XR,AC1,AC2
*
* MONITOR            LC,IR,MAR,XR,AC1,AC2
*
*
* END SIMULATION

```

```

APPB1390
APPB1400
APPB1410
APPB1420
APPB1430
APPB1440
APPB1450
APPB1460
APPB1470
APPB1480
APPB1490
APPB1500
APPB1510
APPB1520
APPB1530
APPB1540
APPB1550
APPB1560
APPB1570
APPB1580
APPB1590
APPB1600
APPB1610
APPB1620
APPB1630
APPB1640
APPB1650
APPB1660

```

Appendix 3. The Syntax Analyzer

INITIALIZATION_STMT	ENTRY	RETURNS(BIT(1)),	APPC1390
MONITOR_STMT	ENTRY	RETURNS(BIT(1)),	APPC1400
DECLARATION_STMT	ENTRY	RETURNS(BIT(1)),	APPC1410
REGISTER_DCL	ENTRY	RETURNS(BIT(1)),	APPC1420
REGISTER_ID	ENTRY	RETURNS(BIT(1)),	APPC1430
SUBREGISTER_DCL	ENTRY	RETURNS(BIT(1)),	APPC1440
SREXP	ENTRY	RETURNS(BIT(1)),	APPC1450
MEMORY_DCL	ENTRY	RETURNS(BIT(1)),	APPC1460
MEMEXP	ENTRY	RETURNS(BIT(1)),	APPC1470
MEMLHS	ENTRY	RETURNS(BIT(1)),	APPC1480
MEMRHS	ENTRY	RETURNS(BIT(1)),	APPC1490
TERMINAL_DCL	ENTRY	RETURNS(BIT(1)),	APPC1500
CONSTANT_DCL	ENTRY	RETURNS(BIT(1)),	APPC1510
CONSEXP	ENTRY	RETURNS(BIT(1)),	APPC1520
OPERATION_DCL	ENTRY	RETURNS(BIT(1)),	APPC1530
COMPOUND_STMT	ENTRY	RETURNS(BIT(1)),	APPC1540
SIMPLE_STMT	ENTRY	RETURNS(BIT(1)),	APPC1550
DO_STMT	ENTRY	RETURNS(BIT(1)),	APPC1560
IF_STMT	ENTRY	RETURNS(BIT(1)),	APPC1570
CONDITION	ENTRY	RETURNS(BIT(1)),	APPC1580
BOOLEAN_TERM	ENTRY	RETURNS(BIT(1)),	APPC1590
BOOLEAN_FACTOR	ENTRY	RETURNS(BIT(1)),	APPC1600
BOOLEAN_SEC	ENTRY	RETURNS(BIT(1)),	APPC1610
BOOLEAN_PRIM	ENTRY	RETURNS(BIT(1)),	APPC1620
RELATION	ENTRY	RETURNS(BIT(1)),	APPC1630
REL_OP	ENTRY	RETURNS(BIT(1)),	APPC1640
SAE	ENTRY	RETURNS(BIT(1)),	APPC1650
LOGICAL_VALUE	ENTRY	RETURNS(BIT(1)),	APPC1660
GO_TO_STMT	ENTRY	RETURNS(BIT(1)),	APPC1670
TRANSFER_STMT	ENTRY	RETURNS(BIT(1)),	APPC1680
DEST	ENTRY	RETURNS(BIT(1)),	APPC1690
UNARY_EXP	ENTRY	RETURNS(BIT(1)),	APPC1700
UNARY_OP	ENTRY	RETURNS(BIT(1)),	APPC1710
UNARY_SOURCE	ENTRY	RETURNS(BIT(1)),	APPC1720
BINARY_EXP	ENTRY	RETURNS(BIT(1)),	APPC1730
BINARY_OP	ENTRY	RETURNS(BIT(1)),	APPC1740
OP1	ENTRY	RETURNS(BIT(1)),	APPC1750
OP2	ENTRY	RETURNS(BIT(1)),	APPC1760

```

    TERMINAL_STMT      ENTRY      RETURNS( BIT(1) ),      APPC1770
    TERMINAL           ENTRY      RETURNS( BIT(1) ),      APPC1780
    LABEL              ENTRY      RETURNS( BIT(1) ),      APPC1790
    TERM               ENTRY      RETURNS( BIT(1) ),      APPC1800
    DEF1               ENTRY      RETURNS( BIT(1) ),      APPC1810
    DEF2               ENTRY      RETURNS( BIT(1) ),      APPC1820
    BOP                ENTRY      RETURNS( BIT(1) ),      APPC1830
    DUMMY              FIXED BINARY(15); /* HAS THE ';' TO END DCL'S */ APPC1840
/* THIS IS THE END OF THE DECLARATIONS */ APPC1850
                                                    APPC1860
                                                    APPC1870
ON CONDITION(EOC) /* END-OF-CARD CONDITION */ APPC1880
  BEGIN; L1: GET EDIT( CARD ) ( A(80) ); APPC1890
          IF C(1)='*' THEN DO; PUT EDIT(CARD)(COL(50),A(80)); APPC1900
          GO TO L1; APPC1910
          END; APPC1920
          J = J + 1; /* INCREMENT STATEMENT COUNTER */ APPC1930
          PUT EDIT( J,CARD ) ( COL(44),F(3),COL(50),A(80) ); APPC1940
          I = 1; /* RESET TEXT POINTER */ APPC1950
END; APPC1960
                                                    APPC1970
          /* THE MAIN PART OF THE PROGRAM FOLLOWS IMMEDIATELY. */ APPC1980
                                                    APPC1990
ON ENDFILE( SYSIN ) GO TO FINIS; APPC2000
PUT EDIT( 'RUN OF THE BAPSIM ANALYZER ' ) (COL(60),A); /* HEADING */ APPC2010
PUT SKIP(5); APPC2020
SIGNAL CONDITION( EOC ); /* GET THE FIRST CARD */ APPC2030
                                                    APPC2040
                                                    APPC2050
IF PROGRAM THEN PUT EDIT('NORMAL ANALYZER EXIT')(SKIP(2),COL(10),A); APPC2060
ELSE PUT EDIT('NO PROGRAM SUPPLIED')(SKIP(2),COL(10),A); APPC2070
APPCC2080
APPCC2090
GO TO DONE; APPC2100
APPCC2110
APPCC2120
APPCC2130
APPCC2140

```

```

/* THE PROCEDURES APPEAR BELOW - PRIMITIVE PROCEDURES FIRST */
LITERAL: PROC( S ) RETURNS( BIT(1) );
DCL S CHAR(*);
IF I>80 THEN SIGNAL CONDITION( EOC );
DO WHILE( C(I)=' ' ); I = I + 1;
IF I>80 THEN SIGNAL CONDITION( EOC );
END;
IF I+LENGTH(S)>81 THEN RETURN( '0'B );
IF SUBSTR(CARD,I,LENGTH(S))=S THEN DO;
I = I + LENGTH(S); RETURN( '1'B ); END;
RETURN( '0'B );
END LITERAL;

ID: PROC RETURNS( BIT(1) );
DCL J FIXED BINARY(31);
IF I>80 THEN SIGNAL CONDITION( EOC );
DO WHILE( C(I)=' ' ); I = I + 1;
IF I>80 THEN SIGNAL CONDITION( EOC );
END;
IF C(I)>'Z' | C(I)<'A' THEN RETURN( '0'B );
DO J=I+1 BY 1 WHILE( C(J)>='A' ); END;
SYMBOL = SUBSTR( CARD,I,J-I ); I = J;
IF LENGTH(SYMBOL)>8 THEN SYMBOL=SUBSTR(SYMBOL,1,4)||
SUBSTR(SYMBOL,LENGTH(SYMBOL)-3,4);
RETURN( '1'B );
END ID;

INT: PROC RETURNS( BIT(1) );
DCL J FIXED BINARY(31);
IF I>80 THEN SIGNAL CONDITION( EOC );
DO WHILE( C(I)=' ' ); I = I + 1;
IF I>80 THEN SIGNAL CONDITION( EOC );
END;
IF C(I)>='0' THEN DO;

```

APPC2150
APPC2160
APPC2170
APPC2180
APPC2190
APPC2200
APPC2210
APPC2220
APPC2230
APPC2240
APPC2250
APPC2260
APPC2270
APPC2280
APPC2290
APPC2300
APPC2310
APPC2320
APPC2330
APPC2340
APPC2350
APPC2360
APPC2370
APPC2380
APPC2390
APPC2400
APPC2410
APPC2420
APPC2430
APPC2440
APPC2450
APPC2460
APPC2470
APPC2480
APPC2490
APPC2500
APPC2510
APPC2520

```

DO J=I+1 BY 1 WHILE( C(J)>='0' ); END;
SYMBOL = SUBSTR( CARD,I,J-I ); I = J;
RETURN( '1'B );          END;
RETURN( '0'B );
END INT;

ERROR: PROC;
PUT EDIT( '*** ERROR AT $ ***, '$' )(COL(1),A,COL(I+49),A);
PUT EDIT( 'RECOVERY NOT POSSIBLE -- COMPILATION ABORTED' )
( SKIP(2),COL(1),A );
CLOSE FILE( SYSPRINT );
STOP;
END ERROR;

/* THE NON-PRIMITIVE PROCEDURES APPEAR NEXT */

PROGRAM: PROC                      RETURNS ( BIT(1) );
IF ~SIMULATION_BLOCK THEN RETURN('0'B);
RETURN ( '1'B );
END PROGRAM;

SIMULATION_BLOCK: PROC             RETURNS( BIT(1) );
IF ~LITERAL('SIMULATION') THEN RETURN('0'B);
IF ~STRUCTURE_BLOCK THEN CALL ERROR;
IF ~BEHAVIOR_BLOCK THEN CALL ERROR;
IF ~IN_OUT_BLOCK THEN CALL ERROR;
IF ~LITERAL( 'END' ) THEN CALL ERROR;
IF ~LITERAL( 'SIMULATION' ) THEN CALL ERROR;
RETURN( '1'B );
END SIMULATION_BLOCK;

STRUCTURE_BLOCK:PROC               RETURNS( BIT(1) );
IF ~LITERAL('DECLARE') THEN RETURN( '0'B );

```

```

APPC2530
APPC2540
APPC2550
APPC2560
APPC2570
APPC2580
APPC2590
APPC2600
APPC2610
APPC2620
APPC2630
APPC2640
APPC2650
APPC2660
APPC2670
APPC2680
APPC2690
APPC2700
APPC2710
APPC2720
APPC2730
APPC2740
APPC2750
APPC2760
APPC2770
APPC2780
APPC2790
APPC2800
APPC2810
APPC2820
APPC2830
APPC2840
APPC2850
APPC2860
APPC2870
APPC2880
APPC2890
APPC2900

```

```

IF ¬DECLARATION_STMT THEN CALL ERROR;
DO WHILE(LITERAL(';',') );
    IF ¬DECLARATION_STMT THEN CALL ERROR;
    END;
IF ¬LITERAL( 'END' ) THEN CALL ERROR;
IF ¬LITERAL( 'DECLARE' ) THEN CALL ERROR;
RETURN( '1'B );
END STRUCTURE_BLOCK;

BEHAVIOR_BLOCK: PROC RETURNS( BIT(1) );
    IF ¬FETCH_BLOCK THEN RETURN( '0'B );
    IF ¬DECODE_BLOCK THEN CALL ERROR;
    IF ¬EXECUTE_BLOCK THEN CALL ERROR;
    RETURN( '1'B );
END BEHAVIOR_BLOCK;

FETCH_BLOCK: PROC RETURNS( BIT(1) );
    IF ¬LITERAL('FETCH') THEN RETURN( '0'B );
    IF ¬SIMPLE_SEQUENCE THEN CALL ERROR;
    IF ¬LITERAL( 'END' ) THEN CALL ERROR;
    IF ¬LITERAL( 'FETCH' ) THEN CALL ERROR;
    RETURN( '1'B );
END FETCH_BLOCK;

SIMPLE_SEQUENCE: PROC RETURNS( BIT(1) );
    IF ¬SIMPLE_STMT THEN RETURN( '0'B );
    DO WHILE( LITERAL(';',') );
        IF ¬SIMPLE_STMT THEN CALL ERROR;
        END;
    RETURN( '1'B );
END SIMPLE_SEQUENCE;

DECODE_BLOCK: PROC RETURNS( BIT(1) );
    IF ¬LITERAL('DECODE') THEN RETURN( '0'B );
    IF ¬SIMPLE_SEQUENCE THEN CALL ERROR;
    IF ¬LITERAL( 'END' ) THEN CALL ERROR;
    IF ¬LITERAL( 'DECODE' ) THEN CALL ERROR;
    RETURN( '1'B );

```

```

APPC2910
APPC2920
APPC2930
APPC2940
APPC2950
APPC2960
APPC2970
APPC2980
APPC2990
APPC3000
APPC3010
APPC3020
APPC3030
APPC3040
APPC3050
APPC3060
APPC3070
APPC3080
APPC3090
APPC3100
APPC3110
APPC3120
APPC3130
APPC3140
APPC3150
APPC3160
APPC3170
APPC3180
APPC3190
APPC3200
APPC3210
APPC3220
APPC3230
APPC3240
APPC3250
APPC3260
APPC3270
APPC3280

```

```

END DECODE_BLOCK;

EXECUTE_BLOCK: PROC                                RETURNS( BIT(1) );
  IF ¬LITERAL( 'EXECUTE' ) THEN RETURN( '0'B );
  IF ¬COMPOUND_STMT THEN CALL ERROR;
  DO WHILE( LITERAL( ';' ) );
    IF ¬COMPOUND_STMT THEN CALL ERROR;
    END;
  IF ¬LITERAL( 'END' ) THEN CALL ERROR;
  IF ¬LITERAL( 'EXECUTE' ) THEN CALL ERROR;
  RETURN( '1'B );
END EXECUTE_BLOCK;

IN_OUT_BLOCK: PROC                                RETURNS( BIT(1) );
  IF ¬INITIALIZATION_STMT THEN RETURN( '0'B );
  IF ¬MONITOR_STMT THEN CALL ERROR;
  RETURN( '1'B );
END IN_OUT_BLOCK;

INITIALIZATION_STMT: PROC                          RETURNS( BIT(1) );
  IF ¬LITERAL( 'INITIALIZE' ) THEN RETURN( '0'B );
  IF ¬ID THEN CALL ERROR;
  DO WHILE( LITERAL( ',' ) );
    IF ¬ID THEN CALL ERROR;
    END;
  RETURN( '1'B );
END INITIALIZATION_STMT;

MONITOR_STMT: PROC                                 RETURNS( BIT(1) );
  IF ¬LITERAL( 'MONITOR' ) THEN RETURN( '0'B );
  IF ¬ID THEN CALL ERROR;
  DO WHILE( LITERAL( ',' ) );
    IF ¬ID THEN CALL ERROR;
    END;
  RETURN( '1'B );
END MONITOR_STMT;

```

```

APPC329C
APPC3300
APPC3310
APPC3320
APPC3330
APPC3340
APPC3350
APPC3360
APPC3370
APPC3380
APPC3390
APPC3400
APPC3410
APPC3420
APPC3430
APPC3440
APPC3450
APPC3460
APPC3470
APPC3480
APPC3490
APPC3500
APPC3510
APPC3520
APPC3530
APPC3540
APPC3550
APPC3560
APPC3570
APPC3580
APPC3590
APPC3600
APPC3610
APPC3620
APPC3630
APPC3640
APPC3650
APPC3660

```

DECLARATION_STMT: PROC	RETURNS(BIT(1));	APPC3670
IF REGISTER_DCL	THEN GO TO L1;	APPC3680
IF SUBREGISTER_DCL	THEN GO TO L1;	APPC3690
IF MEMORY_DCL	THEN GO TO L1;	APPC3700
IF TERMINAL_DCL	THEN GO TO L1;	APPC3710
IF CONSTANT_DCL	THEN GO TO L1;	APPC3720
IF OPERATION_DCL	THEN GO TO L1;	APPC3730
RETURN('0'B);		APPC3740
L1: RETURN('1'B);		APPC3750
END DECLARATION_STMT;		APPC3760
REGISTER_DCL: PROC	RETURNS(BIT(1));	APPC3770
IF ~LITERAL('REGISTER')	THEN RETURN('0'B);	APPC3780
IF ~REGISTER_ID	THEN CALL ERROR;	APPC3790
DO WHILE(LITERAL(','));		APPC3800
IF ~REGISTER_ID	THEN CALL ERROR;	APPC3810
END;		APPC3820
RETURN('1'B);		APPC3830
END REGISTER_DCL;		APPC3840
REGISTER_ID: PROC	RETURNS(BIT(1));	APPC3850
IF ~ID	THEN RETURN('0'B);	APPC3860
IF ~LITERAL('(')	THEN DO;	APPC3870
RETURN('1'B);	END;	APPC3880
IF ~INT	THEN CALL ERROR;	APPC3890
IF ~LITERAL('-')	THEN CALL ERROR;	APPC3900
IF ~INT	THEN CALL ERROR;	APPC3910
IF ~LITERAL(')')	THEN CALL ERROR;	APPC3920
RETURN('1'B);		APPC3930
END REGISTER_ID;		APPC3940
SUBREGISTER_DCL: PROC	RETURNS(BIT(1));	APPC3950
IF ~LITERAL('SUBREGISTER')	THEN RETURN('0'B);	APPC3960
IF ~SREXP	THEN CALL ERROR;	APPC3970
DO WHILE(LITERAL(','));		APPC3980
IF ~SREXP	THEN CALL ERROR;	APPC3990
END;		APPC4000
RETURN('1'B);		APPC4010
		APPC4020
		APPC4030
		APPC4040

END SUBREGISTER_DCL;

```
SREXP: PROC RETURNS( BIT(1) );
  IF ~ID THEN RETURN( '0'B );
  IF ~LITERAL( '(' ) THEN CALL ERROR;
  IF ~ID THEN CALL ERROR;
  IF ~LITERAL( ')' ) THEN CALL ERROR;
  IF ~LITERAL( '=' ) THEN CALL ERROR;
  IF ~ID THEN CALL ERROR;
  IF ~LITERAL( '(' ) THEN CALL ERROR;
  IF ~INT THEN CALL ERROR;
  IF LITERAL( ')' ) THEN DO; RETURN( '1'B ); END;
  IF ~LITERAL( '-' ) THEN CALL ERROR;
  IF ~INT THEN CALL ERROR;
  IF ~LITERAL( ')' ) THEN CALL ERROR;
  RETURN( '1'B );
END SREXP;
```

```
MEMORY_DCL: PROC RETURNS( BIT(1) );
  IF ~LITERAL( 'MEMORY' ) THEN RETURN( '0'B );
  IF ~MEMEXP THEN CALL ERROR;
  DO WHILE( LITERAL( ',', ',' ) );
    IF ~MEMEXP THEN CALL ERROR;
  END;
  RETURN( '1'B );
END MEMORY_DCL;
```

```
MEMEXP: PROC RETURNS( BIT(1) );
  IF ~MEMLHS THEN RETURN( '0'B );
  IF ~LITERAL( '=' ) THEN CALL ERROR;
  IF ~MEMRHS THEN CALL ERROR;
  RETURN( '1'B );
END MEMEXP;
```

```
MEMLHS: PROC RETURNS( BIT(1) );
  IF ~ID THEN RETURN( '0'B );
  IF ~LITERAL( '(' ) THEN CALL ERROR;
  IF ~ID THEN CALL ERROR;
```

APPC4050
APPC4060
APPC4070
APPC4080
APPC4090
APPC4100
APPC4110
APPC4120
APPC4130
APPC4140
APPC4150
APPC4160
APPC4170
APPC4180
APPC4190
APPC4200
APPC4210
APPC4220
APPC4230
APPC4240
APPC4250
APPC4260
APPC4270
APPC4280
APPC4290
APPC4300
APPC4310
APPC4320
APPC4330
APPC4340
APPC4350
APPC4360
APPC4370
APPC4380
APPC4390
APPC4400
APPC4410
APPC4420

```

        IF LITERAL( ' )' ) THEN RETURN( '1'B );
        IF ~LITERAL( '(' ) THEN CALL ERROR;
        IF ~ID THEN CALL ERROR;
        IF ~LITERAL( ')' ) THEN CALL ERROR;
        IF ~LITERAL( ']' ) THEN CALL ERROR;
        RETURN( '1'B );
END MEMLHS;

MEMRHS: PROC                                RETURNS( BIT(1) );
    IF ~ID THEN RETURN( '0'B );
    IF ~LITERAL( '(' ) THEN CALL ERROR;
    IF ~INT THEN CALL ERROR;
    IF ~LITERAL( '--' ) THEN CALL ERROR;
    IF ~INT THEN CALL ERROR;
    IF ~LITERAL( ',' ) THEN CALL ERROR;
    IF ~INT THEN CALL ERROR;
    IF ~LITERAL( '--' ) THEN CALL ERROR;
    IF ~INT THEN CALL ERROR;
    IF ~LITERAL( ']' ) THEN CALL ERROR;
    RETURN( '1'B );
END MEMRHS;

TERMINAL_DCL: PROC                          RETURNS( BIT(1) );
    IF ~LITERAL( 'TERMINAL' ) THEN RETURN( '0'B );
    IF ~REGISTER_ID THEN CALL ERROR;
    DO WHILE( LITERAL( ',' ) );
        IF ~REGISTER_ID THEN CALL ERROR;
    END;
    RETURN( '1'B );
END TERMINAL_DCL;

CONSTANT_DCL: PROC                          RETURNS( BIT(1) );
    IF ~LITERAL( 'CONSTANT' ) THEN RETURN( '0'B );
    IF ~CONSEXP THEN CALL ERROR;
    DO WHILE( LITERAL( ',' ) );
        IF ~CONSEXP THEN CALL ERROR;
    END;
    RETURN( '1'B );

```

```

APPC4430
APPC4440
APPC4450
APPC4460
APPC4470
APPC4480
APPC4490
APPC4500
APPC4510
APPC4520
APPC4530
APPC4540
APPC4550
APPC4560
APPC4570
APPC4580
APPC4590
APPC4600
APPC4610
APPC4620
APPC4630
APPC4640
APPC4650
APPC4660
APPC4670
APPC4680
APPC4690
APPC4700
APPC4710
APPC4720
APPC4730
APPC4740
APPC4750
APPC4760
APPC4770
APPC4780
APPC4790
APPC4800

```

```

END CONSTANT_DCL;

CONSEXP: PROC RETURNS( BIT(1) );
  IF ~ID THEN RETURN( '0'B );
  IF ~LITERAL( '=' ) THEN CALL ERROR;
  IF ~INT THEN CALL ERROR;
  RETURN( '1'B );
END CONSEXP;

OPERATION_DCL: PROC RETURNS( BIT(1) );
  IF ~LITERAL( 'OPERATION' ) THEN RETURN( '0'B );
  IF ~ID THEN CALL ERROR;
  DO WHILE( LITERAL( ',', ',' ) );
    IF ~ID THEN CALL ERROR;
  END;
  RETURN( '1'B );
END OPERATION_DCL;

COMPOUND_STMT: PROC RETURNS( BIT(1) );
  IF ~ID THEN RETURN( '0'B );
  IF ~LITERAL( ':' ) THEN CALL ERROR;
  IF ~SIMPLE_STMT THEN CALL ERROR;
  DO WHILE( LITERAL( ',', ',' ) );
    IF ~SIMPLE_STMT THEN CALL ERROR;
  END;
  RETURN( '1'B );
END COMPOUND_STMT;

SIMPLE_STMT: PROC RETURNS( BIT(1) ) RECURSIVE;
  IF DO_STMT THEN GO TO L1;
  IF IF_STMT THEN GO TO L1;
  IF GO_TO_STMT THEN GO TO L1;
  IF TRANSFER_STMT THEN GO TO L1;
  IF TERMINAL_STMT THEN GO TO L1;
  RETURN( '0'B );
L1: IF ~LITERAL( '<' ) THEN GO TO L2;
  IF ~INT THEN CALL ERROR;
  IF ~LITERAL( '>' ) THEN CALL ERROR;

```

```

APPC4810
APPC4820
APPC483C
APPC4840
APPC4850
APPC4860
APPC4870
APPC4880
APPC4890
APPC4900
APPC4910
APPC4920
APPC4930
APPC4940
APPC4950
APPC4960
APPC4970
APPC4980
APPC4990
APPC5000
APPC5010
APPC5020
APPC5030
APPC5040
APPC5050
APPC5060
APPC5070
APPC5080
APPC5090
APPC5100
APPC5110
APPC5120
APPC5130
APPC5140
APPC5150
APPC5160
APPC5170
APPC5180

```

```

L2: RETURN( '1'B );
END SIMPLE_STMT;

DO_STMT: PROC RETURNS( BIT(1) ) RECURSIVE;
  IF ¬LITERAL('DO') THEN RETURN( '0'B );
  IF ¬ID THEN CALL ERROR;
  RETURN( '1'B );
END DO_STMT;

IF_STMT: PROC RETURNS( BIT(1) ) RECURSIVE;
  IF ¬LITERAL('IF') THEN RETURN( '0'B );
  IF ¬CONDITION THEN CALL ERROR;
  IF ¬LITERAL('THEN') THEN CALL ERROR;
  IF ¬SIMPLE_STMT THEN CALL ERROR;
  RETURN( '1'B );
END IF_STMT;

CONDITION: PROC RETURNS( BIT(1) );
  IF ¬LITERAL( '(' ) THEN RETURN( '0'B );
  IF ¬BOOLEAN_TERM THEN CALL ERROR;
  IF ¬LITERAL( ')' ) THEN CALL ERROR;
  RETURN( '1'B );
END CONDITION;

BOOLEAN_TERM: PROC RETURNS( BIT(1) ) RECURSIVE;
  IF ¬BOOLEAN_FACTOR THEN RETURN( '0'B );
  DO WHILE( LITERAL( '.OR.' ) );
    IF ¬BOOLEAN_FACTOR THEN CALL ERROR;
  END;
  RETURN( '1'B );
END BOOLEAN_TERM;

BOOLEAN_FACTOR: PROC RETURNS( BIT(1) ) RECURSIVE;
  IF ¬BOOLEAN_SEC THEN RETURN( '0'B );
  DO WHILE( LITERAL( '.AND.' ) );
    IF ¬BOOLEAN_SEC THEN CALL ERROR;
  END;
  RETURN( '1'B );

```

```

APPC5190
APPC5200
APPC5210
APPC5220
APPC5230
APPC5240
APPC5250
APPC5260
APPC5270
APPC5280
APPC5290
APPC5300
APPC5310
APPC5320
APPC5330
APPC5340
APPC5350
APPC5360
APPC5370
APPC5380
APPC5390
APPC5400
APPC5410
APPC5420
APPC5430
APPC5440
APPC5450
APPC5460
APPC5470
APPC5480
APPC5490
APPC5500
APPC5510
APPC5520
APPC5530
APPC5540
APPC5550
APPC5560

```

END BOOLEAN_FACTOR;	APPC5570
BOOLEAN_SEC: PROC RETURNS(BIT(1)) RECURSIVE;	APPC5580
IF LITERAL('.NOT.') THEN ;	APPC5590
IF BOOLEAN_PRIM THEN GO TO L1;	APPC5600
RETURN('0'B);	APPC5610
L1: RETURN('1'B);	APPC5620
END BOOLEAN_SEC;	APPC5630
BOOLEAN_PRIM: PROC RETURNS(BIT(1)) RECURSIVE;	APPC5640
IF LOGICAL_VALUE THEN RETURN('1'B);	APPC5650
IF RELATION THEN RETURN('1'B);	APPC5660
IF ¬LITERAL('(') THEN RETURN('0'B);	APPC5670
IF ¬BOOLEAN_TERM THEN CALL ERROR;	APPC5680
IF ¬LITERAL(')') THEN CALL ERROR;	APPC5690
RETURN('1'B);	APPC5700
END BOOLEAN_PRIM;	APPC5710
RELATION: PROC RETURNS(BIT(1)) RECURSIVE;	APPC5720
IF ¬SAE THEN RETURN('0'B);	APPC5730
IF ¬REL_OP THEN RETURN('1'B);	APPC5740
IF ¬SAE THEN CALL ERROR;	APPC5750
RETURN('1'B);	APPC5760
END RELATION;	APPC5770
REL_OP: PROC RETURNS(BIT(1)) RECURSIVE;	APPC5780
IF LITERAL('.EQ.') THEN GO TO L1;	APPC5790
IF LITERAL('.NE.') THEN GO TO L1;	APPC5800
IF LITERAL('.LT.') THEN GO TO L1;	APPC5810
IF LITERAL('.GT.') THEN GO TO L1;	APPC5820
IF LITERAL('.LE.') THEN GO TO L1;	APPC5830
IF LITERAL('.GE.') THEN GO TO L1;	APPC5840
RETURN('0'B);	APPC5850
L1: RETURN('1'B);	APPC5860
END REL_OP;	APPC5870
SAE: PROC RETURNS(BIT(1)) RECURSIVE;	APPC5880
IF INT THEN GO TO L1;	APPC5890
	APPC5900
	APPC5910
	APPC5920
	APPC5930
	APPC5940

IF DEST THEN GO TO L1;	APPC5950
RETURN('0'B);	APPC5960
L1: RETURN('1'B);	APPC5970
END SAE;	APPC5980
LOGICAL_VALUE: PROC	APPC5990
IF LITERAL('.TRUE.') THEN GO TO L1;	APPC6000
IF LITERAL('.FALSE.') THEN GO TO L1;	APPC6010
RETURN('0'B);	APPC6020
L1: RETURN('1'B);	APPC6030
END LOGICAL_VALUE;	APPC6040
GO_TO_STMT: PROC	APPC6050
IF NOT LITERAL('GO') THEN RETURN('0'B);	APPC6060
IF NOT LITERAL('TO') THEN CALL ERROR;	APPC6070
IF NOT ID THEN CALL ERROR;	APPC6080
RETURN('1'B);	APPC6090
END GO_TO_STMT;	APPC6100
TRANSFER_STMT: PROC	APPC6110
DCL BACKUP	APPC6120
BACKUP = I;	APPC6130
IF NOT DEST THEN RETURN('0'B);	APPC6140
IF NOT LITERAL('<') THEN DO;	APPC6150
I = BACKUP;	APPC6160
RETURN('0'B);	APPC6170
END;	APPC6180
IF INT THEN GO TO DONE;	APPC6190
IF UNARY_EXP THEN GO TO DONE;	APPC6200
IF BINARY_EXP THEN GO TO DONE;	APPC6210
CALL ERROR;	APPC6220
DONE: RETURN('1'B);	APPC6230
END TRANSFER_STMT;	APPC6240
DEST: PROC	APPC6250
IF NOT ID THEN RETURN('0'B);	APPC6260
IF NOT LITERAL('(') THEN GO TO L1;	APPC6270
IF ID THEN DO;	APPC6280
IF LITERAL(')') THEN DO;	APPC6290
	APPC6300
	APPC6310
	APPC6320

GO TO L1;	END;	APPC6330
	ELSE CALL ERROR;	APPC6340
END;		APPC6350
IF ~INT THEN CALL ERROR;		APPC6360
IF LITERAL(')') THEN GO TO L1;		APPC6370
IF ~LITERAL(' - ') THEN CALL ERROR;		APPC6380
IF ~INT THEN CALL ERROR;		APPC6390
IF ~LITERAL(')') THEN CALL ERROR;		APPC6400
L1: RETURN('1'B);		APPC6410
END DEST;		APPC6420
		APPC6430
UNARY_EXP: PROC	RETURNS(BIT(1));	APPC6440
IF ~UNARY_OP THEN RETURN('0'B);		APPC6450
IF ~UNARY_SOURCE THEN CALL ERROR;		APPC6460
RETURN('1'B);		APPC6470
END UNARY_EXP;		APPC6480
		APPC6490
UNARY_OP: PROC	RETURNS(BIT(1));	APPC6500
IF LITERAL(' .AND. ') THEN GO TO L1;		APPC6510
IF LITERAL(' .OR. ') THEN GO TO L1;		APPC6520
IF LITERAL(' .NOT. ') THEN GO TO L1;		APPC6530
IF LITERAL(' .SHL. ') THEN GO TO L1;		APPC6540
IF LITERAL(' .SHR. ') THEN GO TO L1;		APPC6550
IF LITERAL(' .CIRL. ') THEN GO TO L1;		APPC6560
IF LITERAL(' .CIRR. ') THEN GO TO L1;		APPC6570
RETURN('0'B);		APPC6580
L1: RETURN('1'B);		APPC6590
END UNARY_OP;		APPC6600
		APPC6610
UNARY_SOURCE: PROC	RETURNS(BIT(1));	APPC6620
IF ~ID THEN RETURN('0'B);		APPC6630
IF ~LITERAL('(') THEN GO TO L1;		APPC6640
IF ID THEN DO;		APPC6650
IF LITERAL(') ') THEN DO;		APPC6660
GO TO L1;	END;	APPC6670
	ELSE CALL ERROR;	APPC6680
END;		APPC6690
IF ~INT THEN CALL ERROR;		APPC6700

IF LITERAL(')')	THEN GO TO L1;	APPC6710
IF ¬LITERAL('−')	THEN CALL ERROR;	APPC6720
IF ¬INT	THEN CALL ERROR;	APPC6730
IF ¬LITERAL(')')	THEN CALL ERROR;	APPC6740
L1: RETURN('1'B);		APPC6750
END UNARY_SOURCE;		APPC6760
BINARY_EXP: PROC	RETURNS(BIT(1));	APPC6770
IF ¬OP1	THEN RETURN('0'B);	APPC6780
IF ¬BINARY_OP	THEN GO TO L1;	APPC6790
IF OP2	THEN GO TO L1;	APPC6800
IF INT	THEN GO TO L1;	APPC6810
CALL ERROR;		APPC6820
L1: RETURN('1'B);		APPC6830
END BINARY_EXP;		APPC6840
BINARY_OP: PROC	RETURNS(BIT(1));	APPC6850
IF LITERAL('.ADD.')	THEN GO TO L1;	APPC6860
IF LITERAL('.SUB.')	THEN GO TO L1;	APPC6880
IF LITERAL('.OR.')	THEN GO TO L1;	APPC6890
IF LITERAL('.AND.')	THEN GO TO L1;	APPC6900
IF LITERAL('.XOR.')	THEN GO TO L1;	APPC6910
IF LITERAL('.SHL.')	THEN GO TO L1;	APPC6920
IF LITERAL('.SHR.')	THEN GO TO L1;	APPC6930
IF LITERAL('.CNT.')	THEN GO TO L1;	APPC6940
RETURN('0'B);		APPC6950
L1: RETURN('1'B);		APPC6960
END BINARY_OP;		APPC6970
OP1: PROC	RETURNS(BIT(1));	APPC6980
IF ¬ID	THEN RETURN('0'B);	APPC6990
IF ¬LITERAL('(')	THEN GO TO L1;	APPC7000
IF ID	THEN DO;	APPC7010
IF LITERAL(')')	THEN DO;	APPC7020
GO TO L1;	END;	APPC7030
ELSE CALL ERROR;		APPC7040
END;		APPC7050
IF ¬INT	THEN CALL ERROR;	APPC7060
		APPC7070
		APPC7080

IF LITERAL(')')	THEN GO TO L1;	APPC7090
IF ~LITERAL('~-')	THEN CALL ERROR;	APPC7100
IF ~INT THEN	CALL ERROR;	APPC7110
IF ~LITERAL(')')	THEN CALL ERROR;	APPC7120
L1: RETURN('1'B);		APPC7130
END OP1;		APPC7140
OP2: PROC	RETURNS(BIT(1));	APPC7150
IF ~ID THEN	RETURN('0'B);	APPC7160
IF ~LITERAL('(')	THEN GO TO L1;	APPC7170
IF ID THEN	DO;	APPC7180
IF LITERAL(')') THEN DO;	APPC7190
GO TO L1;	END;	APPC7200
ELSE	CALL ERROR;	APPC7210
END;		APPC7220
IF ~INT THEN	CALL ERROR;	APPC7230
IF LITERAL(')')	THEN GO TO L1;	APPC7240
IF ~LITERAL('~-')	THEN CALL ERROR;	APPC7250
IF ~INT THEN	CALL ERROR;	APPC7260
IF ~LITERAL(')')	THEN CALL ERROR;	APPC7270
L1: RETURN('1'B);		APPC7280
END OP2;		APPC7290
TERMINAL_STMT: PROC	RETURNS(BIT(1));	APPC7300
DCL	BACKUP	FIXED BINARY(31);
BACKUP = I;	/*	SAVE PTR TO TEXT POSITION */
IF ~TERMINAL THEN	RETURN('0'B);	APPC7310
IF ~LITERAL(':=')	THEN DO;	APPC7320
I = BACKUP;	/*	BACKUP TEXT PTR */
RETURN('0'B);	END;	APPC7330
IF ~LABEL THEN	CALL ERROR;	APPC7340
RETURN('1'B);		APPC7350
END TERMINAL_STMT;		APPC7360
TERMINAL: PROC	RETURNS(BIT(1));	APPC7370
IF ~ID THEN	RETURN('0'B);	APPC7380
IF ~LITERAL('(')	THEN GO TO L1;	APPC7390
IF ~INT THEN	CALL ERROR;	APPC7400
		APPC7410
		APPC7420
		APPC7430
		APPC7440
		APPC7450
		APPC7460

```

        IF ¬LITERAL( ' )' ) THEN CALL ERROR;
    L1: RETURN( '1'B );
END TERMINAL;

LABEL: PROC                                RETURNS( BIT(1) );
    IF ¬TERM THEN RETURN( '0'B );
    L: IF ¬BOP THEN GO TO L1;
    IF ¬TERM THEN CALL ERROR;
    GO TO L;
    L1: RETURN( '1'B );
END LABEL;

TERM: PROC                                RETURNS( BIT(1) );
    IF DEF1 THEN GO TO L1;
    IF DEF2 THEN GO TO L1;
    RETURN( '0'B );
    L1: RETURN( '1'B );
END TERM;

DEF1: PROC                                RETURNS( BIT(1) );
    IF ¬ID THEN RETURN( '0'B );
    IF ¬LITERAL( '(' ) THEN GO TO L1;
    IF ¬INT THEN CALL ERROR;
    IF ¬LITERAL( ')' ) THEN CALL ERROR;
    L1: RETURN( '1'B );
END DEF1;

DEF2: PROC                                RETURNS( BIT(1) );
    IF ¬LITERAL( '(' ) THEN RETURN( '0'B );
    IF ¬LITERAL( '~' ) THEN CALL ERROR;
    IF ¬DEF1 THEN CALL ERROR;
    IF ¬LITERAL( ')' ) THEN CALL ERROR;
    RETURN( '1'B );
END DEF2;

BOP: PROC                                RETURNS( BIT(1) );
    IF LITERAL( '*' ) THEN GO TO L1;
    IF LITERAL( '+' ) THEN GO TO L1;

```

```

APPC7470
APPC7480
APPC7490
APPC7500
APPC7510
APPC7520
APPC7530
APPC7540
APPC7550
APPC7560
APPC7570
APPC7580
APPC7590
APPC7600
APPC7610
APPC7620
APPC7630
APPC7640
APPC7650
APPC7660
APPC7670
APPC7680
APPC7690
APPC7700
APPC7710
APPC7720
APPC7730
APPC7740
APPC7750
APPC7760
APPC7770
APPC7780
APPC7790
APPC7800
APPC7810
APPC7820
APPC7830
APPC7840

```

```
        RETURN( '0'B );  
L1: RETURN( '1'B );  
END BOP;
```

```
/* THIS IS THE END OF THE INTERNAL PROCEDURES */
```

```
FINIS: PUT EDIT('END OF FILE SENSED ON SYSIN')(SKIP(2),COL(10),A);  
DONE:  
        END COMPILE;
```

```
APPC7850  
APPC7860  
APPC7870  
APPC7880  
APPC7890  
APPC7900  
APPC7910  
APPC7920  
APPC7930  
APPC7940  
APPC7950  
APPC7960  
APPC7970  
APPC7980
```

Appendix 4. The BAPSIM Translator

PROGRAM	ENTRY	RETURNS (BIT(1)),	APPD1078
SIMULATION_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1080
STRUCTURE_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1082
BEHAVIOR_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1084
FETCH_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1086
SIMPLE_SEQUENCE	ENTRY	RETURNS (BIT(1)),	APPD1088
DECODE_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1090
EXECUTE_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1092
IN_OUT_BLOCK	ENTRY	RETURNS (BIT(1)),	APPD1094
INITIALIZATION_STMT	ENTRY	RETURNS (BIT(1)),	APPD1096
MONITOR_STMT	ENTRY	RETURNS (BIT(1)),	APPD1098
DECLARATION_STMT	ENTRY	RETURNS (BIT(1)),	APPD1100
REGISTER_DCL	ENTRY	RETURNS (BIT(1)),	APPD1102
REGISTER_ID	ENTRY	RETURNS (BIT(1)),	APPD1104
SUBREGISTER_DCL	ENTRY	RETURNS (BIT(1)),	APPD1106
SREXP	ENTRY	RETURNS (BIT(1)),	APPD1108
MEMORY_DCL	ENTRY	RETURNS (BIT(1)),	APPD1110
MEMEXP	ENTRY	RETURNS (BIT(1)),	APPD1112
MEMLHS	ENTRY	RETURNS (BIT(1)),	APPD1114
MEMRHS	ENTRY	RETURNS (BIT(1)),	APPD1116
TERMINAL_DCL	ENTRY	RETURNS (BIT(1)),	APPD1118
CONSTANT_DCL	ENTRY	RETURNS (BIT(1)),	APPD1120
CONSEXP	ENTRY	RETURNS (BIT(1)),	APPD1122
OPERATION_DCL	ENTRY	RETURNS (BIT(1)),	APPD1124
COMPOUND_STMT	ENTRY	RETURNS (BIT(1)),	APPD1126
SIMPLE_STMT	ENTRY	RETURNS (BIT(1)),	APPD1128
DO_STMT	ENTRY	RETURNS (BIT(1)),	APPD1130
IF_STMT	ENTRY	RETURNS (BIT(1)),	APPD1132
CONDITION	ENTRY	RETURNS (BIT(1)),	APPD1134
BOOLEAN_TERM	ENTRY	RETURNS (BIT(1)),	APPD1136
BOOLEAN_FACTOR	ENTRY	RETURNS (BIT(1)),	APPD1138
BOOLEAN_SEC	ENTRY	RETURNS (BIT(1)),	APPD1140
BOOLEAN_PRIM	ENTRY	RETURNS (BIT(1)),	APPD1142
RELATION	ENTRY	RETURNS (BIT(1)),	APPD1144
REL_OP	ENTRY	RETURNS (BIT(1)),	APPD1146
SAE	ENTRY	RETURNS (BIT(1)),	APPD1148
LOGICAL_VALUE	ENTRY	RETURNS (BIT(1)),	APPD1150
			APPD1152

```

GO_TO_STMT          ENTRY      RETURNS( BIT(1) ),      APPD1154
TRANSFER_STMT      ENTRY      RETURNS( BIT(1) ),      APPD1156
DEST                ENTRY      RETURNS( BIT(1) ),      APPD1158
UNARY_EXP           ENTRY      RETURNS( BIT(1) ),      APPD1160
UNARY_OP            ENTRY      RETURNS( BIT(1) ),      APPD1162
UNARY_SOURCE        ENTRY      RETURNS( BIT(1) ),      APPD1164
BINARY_EXP          ENTRY      RETURNS( BIT(1) ),      APPD1166
BINARY_OP           ENTRY      RETURNS( BIT(1) ),      APPD1168
OP1                 ENTRY      RETURNS( BIT(1) ),      APPD1170
OP2                 ENTRY      RETURNS( BIT(1) ),      APPD1172
TERMINAL_STMT      ENTRY      RETURNS( BIT(1) ),      APPD1174
TERMINAL            ENTRY      RETURNS( BIT(1) ),      APPD1176
LABEL              ENTRY      RETURNS( BIT(1) ),      APPD1178
TERM                ENTRY      RETURNS( BIT(1) ),      APPD1180
DEF1                ENTRY      RETURNS( BIT(1) ),      APPD1182
DEF2                ENTRY      RETURNS( BIT(1) ),      APPD1184
BOP                 ENTRY      RETURNS( BIT(1) ),      APPD1186
DUMMY              FIXED BINARY(15); /* HAS THE ';' TO END DCL'S */ APPD1188
DECLARE ASSM FILE;
/* THIS IS THE END OF THE DECLARATIONS */ APPD1190
APPD1192
APPD1194
APPD1196
ON CONDITION(EOC) /* END-OF-CARD CONDITION */ APPD1198
BEGIN; L1: GET EDIT( CARD ) ( A(80) ); APPD1200
IF C(1)='*' THEN DO; PUT EDIT(CARD)(COL(50),A(80)); APPD1202
GO TO L1; APPD1204
END; APPD1206
J = J + 1; /* INCREMENT STATEMENT COUNTER */ APPD1208
PUT EDIT( J,CARD ) ( COL(44),F(3),COL(50),A(80) ); APPD1210
I = 1; /* RESET TEXT POINTER */ APPD1212
END; APPD1214
APPD1216
/* THE MAIN PART OF THE PROGRAM FOLLOWS IMMEDIATELY. */ APPD1218
APPD1220
ON ENDFILE( SYSIN ) GO TO FINIS; APPD1222
PUT EDIT( 'RUN OF THE BAPSIM TRANSLATOR' ) (COL(60),A); /* HEADING */ APPD1224
PUT EDIT( '*** GENERATED CODE ***', '*** BAPSIM SOURCE CODE ***' ) APPD1226
(SKIP(4),COL(1),A,COL(50),A); APPD1228

```

```

PUT SKIP(5);
SIGNAL CONDITION( FOC ); /* GET THE FIRST CARD */
OPEN FILE(ASSM) RECORD OUTPUT;

```

```

IF PROGRAM THEN PUT EDIT('NORMAL COMPILER EXIT')(SKIP(2),COL(10),A);
ELSE PUT EDIT('NO PROGRAM SUPPLIED')(SKIP(2),COL(10),A);

```

```

PUT FILE(SYS$PRINT) EDIT /* I/O COUNT FOR DEBUG */
( IO, ' SEMANTIC-ACTION RECORDS WERE GENERATED.' )
( SKIP(2),COL(10),F(4),A(41) );
GO TO DONE;

```

```

/* THE PROCEDURES APPEAR BELOW - PRIMITIVE PROCEDURES FIRST */

```

```

LITERAL: PROC( S ) RETURNS( BIT(1) );
DCL S CHAR(*);
IF I>80 THEN SIGNAL CONDITION( EOC );
DO WHILE( C(I)=' ' ); I = I + 1;
IF I>80 THEN SIGNAL CONDITION( EOC );
END;
IF I+LENGTH(S)>81 THEN RETURN( '0'B );
IF SUBSTR(CARD,I,LENGTH(S))=S THEN DO;
I = I + LENGTH(S); RETURN( '1'B ); END;
RETURN( '0'B );
END LITERAL;

```

```

ID: PROC RETURNS( BIT(1) );
DCL J FIXED BINARY(31);
IF I>80 THEN SIGNAL CONDITION( EOC );
DO WHILE( C(I)=' ' ); I = I + 1;
IF I>80 THEN SIGNAL CONDITION( EOC );

```

```

APPD1230
APPD1232
APPD1234
APPD1236
APPD1238
APPD1240
APPD1242
APPD1244
APPD1246
APPD1248
APPD1250
APPD1252
APPD1254
APPD1256
APPD1258
APPD1260
APPD1262
APPD1264
APPD1266
APPD1268
APPD1270
APPD1272
APPD1274
APPD1276
APPD1278
APPD1280
APPD1282
APPD1284
APPD1286
APPD1288
APPD1290
APPD1292
APPD1294
APPD1296
APPD1298
APPD1300
APPD1302
APPD1304

```

```

        END;
        IF C(I)>'Z' | C(I)<'A' THEN RETURN( '0'B );
        DO J=I+1 BY 1 WHILE( C(J)>='A' ); END;
        SYMBOL = SUBSTR( CARD, I, J-I ); I = J;
IF LENGTH(SYMBOL)>8 THEN SYMBOL=SUBSTR(SYMBOL,1,4)||
        SUBSTR(SYMBOL,LENGTH(SYMBOL)-3,4);
        RETURN( '1'B );
END ID;

INT: PROC          RETURNS( BIT(1) );
DCL      J          FIXED BINARY(31);
IF I>80 THEN SIGNAL CONDITION( EOC );
DO WHILE( C(I)=' ' ); I = I + 1;
IF I>80 THEN SIGNAL CONDITION( EOC );
END;
IF C(I)>='0' THEN DO;
DO J=I+1 BY 1 WHILE( C(J)>='0' ); END;
SYMBOL = SUBSTR( CARD, I, J-I ); I = J;
RETURN( '1'B );      END;
RETURN( '0'B );
END INT;

ERROR: PROC;
PUT EDIT( '*** ERROR AT $ ***, '$' )(COL(1),A,COL(I+49),A);
PUT EDIT( 'RECOVERY NOT POSSIBLE -- COMPILATION ABORTED' )
        ( SKIP(2),COL(1),A );
CLOSE FILE( SYSPRINT );
CLOSE FILE( ASSM );
STOP;
END ERROR;

STAR: PROC          RETURNS( CHAR(32) VARYING );
RETURN( SYMBOL );
END STAR;

```

```

APPD1306
APPD1308
APPD1310
APPD1312
APPD1314
APPD1316
APPD1318
APPD1320
APPD1322
APPD1324
APPD1326
APPD1328
APPD1330
APPD1332
APPD1334
APPD1336
APPD1338
APPD1340
APPD1342
APPD1344
APPD1346
APPD1348
APPD1350
APPD1352
APPD1354
APPD1356
APPD1358
APPD1360
APPD1362
APPD1364
APPD1366
APPD1368
APPD1370
APPD1372
APPD1374
APPD1376
APPD1378
APPD1380

```

```

OUT: PROC( S );
    DCL S CHAR(*);
    DCL KARD CHARACTER(80);
    KARD = S;
    WRITE FILE(ASSM) FROM(KARD);
    PUT EDIT( S ) ( COL(1),A );
    IO = IO + 1; /* I/O COUNTER FOR DEBUG */
END OUT;

```

/* THE NON-PRIMITIVE PROCEDURES APPEAR NEXT */

```

PROGRAM: PROC RETURNS ( BIT(1) );
    IF ~SIMULATION_BLOCK THEN RETURN('0'B);
    RETURN ('1'B);
END PROGRAM;

```

```

SIMULATION_BLOCK: PROC RETURNS( BIT(1) );
    IF ~LITERAL('SIMULATION') THEN RETURN('0'B);
    LINE = ' (SIZE,SUBSCRIPTRANGE,STRINGRANGE):' ;
    CALL OUT(LINE);
    LINE=' B#PS#M: PROC OPTIONS(MAIN);' ; CALL OUT(LINE);
    IF ~STRUCTURE_BLOCK THEN CALL ERROR;
    IF ~BEHAVIOR_BLOCK THEN CALL ERROR;
    IF ~IN_OUT_BLOCK THEN CALL ERROR;
    IF ~LITERAL('END') THEN CALL ERROR;
    IF ~LITERAL('SIMULATION') THEN CALL ERROR;
    LINE = ' STOP: C#CLE = 9999; GO TO D#T;' ; CALL OUT(LINE);
    LINE = ' D#NE: PUT FILE(SYSPRINT) EDIT' ;
    CALL OUT(LINE);
    LINE=' ( ''NORMAL SIMULATOR EXIT -- SUCCESSFUL RUN'' )' ;
    CALL OUT(LINE);
    LINE = ' (SKIP(5),COL(10),A);' ;
    CALL OUT(LINE);

```

```

APPD1382
APPD1384
APPD1386
APPD1388
APPD1390
APPD1392
APPD1394
APPD1396
APPD1398
APPD1400
APPD1402
APPD1404
APPD1406
APPD1408
APPD1410
APPD1412
APPD1414
APPD1416
APPD1418
APPD1420
APPD1422
APPD1424
APPD1426
APPD1428
APPD1430
APPD1432
APPD1434
APPD1436
APPD1438
APPD1440
APPD1442
APPD1444
APPD1446
APPD1448
APPD1450
APPD1452
APPD1454
APPD1456

```

```

        LINE=' END B#PS#M;' ;                CALL OUT(LINE);                APPD1458
        RETURN( '1'B);                        APPD1460
END SIMULATION_BLOCK;                        APPD1462
STRUCTURE_BLOCK: PROC                        RETURNS( BIT(1) );                APPD1464
    IF ~LITERAL('DECLARE') THEN RETURN( '0'B); APPD1466
    IF ~DECLARATION_STMT THEN CALL ERROR;    APPD1468
    DO WHILE(LITERAL(';')) ;                APPD1470
        IF ~DECLARATION_STMT THEN CALL ERROR; APPD1472
        END;                                APPD1474
    IF ~LITERAL( 'END' ) THEN CALL ERROR;    APPD1476
    IF ~LITERAL( 'DECLARE' ) THEN CALL ERROR; APPD1478
    LINE = ' %INCLUDE DCLS;                  /* * * * * * * * * */'; APPD1480
    CALL OUT(LINE);                          APPD1482
    RETURN( '1'B);                            APPD1484
END STRUCTURE_BLOCK;                        APPD1486
BEHAVIOR_BLOCK: PROC                        RETURNS( BIT(1) );                APPD1488
    IF ~FETCH_BLOCK THEN RETURN( '0'B );    APPD1490
    IF ~DECODE_BLOCK THEN CALL ERROR;        APPD1492
    IF ~EXECUTE_BLOCK THEN CALL ERROR;        APPD1494
    RETURN( '1'B );                          APPD1496
END BEHAVIOR_BLOCK;                         APPD1498
FETCH_BLOCK: PROC                          RETURNS( BIT(1) );                APPD1500
    IF ~LITERAL('FETCH') THEN RETURN( '0'B ); APPD1502
    LINE = ' F#TCH:' ;                       CALL OUT(LINE);                APPD1504
    IF ~SIMPLE_SEQUENCE THEN CALL ERROR;     APPD1506
    IF ~LITERAL( 'END' ) THEN CALL ERROR;    APPD1508
    IF ~LITERAL( 'FETCH' ) THEN CALL ERROR;  APPD1510
    LINE = ' GO TO D#COD#;' ;                CALL OUT(LINE);                APPD1512
    RETURN( '1'B );                          APPD1514
END FETCH_BLOCK;                           APPD1516
SIMPLE_SEQUENCE: PROC                      RETURNS( BIT(1) );                APPD1518
    IF ~SIMPLE_STMT THEN RETURN( '0'B );     APPD1520
    DO WHILE( LITERAL( ',' ) );              APPD1522
        IF ~SIMPLE_STMT THEN CALL ERROR;    APPD1524
        APPD1526
        APPD1528
        APPD1530
        APPD1532

```

```

        END;
        RETURN( '1'B );
END SIMPLE_SEQUENCE;

DECODE_BLOCK: PROC                                RETURNS( BIT(1) );
    IF ¬LITERAL('DECODE') THEN RETURN( '0'B );
    LINE = ' D#COD#:' ;                          CALL OUT(LINE);
    IF ¬SIMPLE_SEQUENCE THEN CALL ERROR;
    IF ¬LITERAL( 'END' ) THEN CALL ERROR;
    IF ¬LITERAL( 'DECODE' ) THEN CALL ERROR;
    LINE = ' %INCLUDE ENDCODE;                    /* * * * * * */';
    CALL OUT(LINE);
    RETURN( '1'B );
END DECODE_BLOCK;

EXECUTE_BLOCK: PROC                               RETURNS( BIT(1) );
    IF ¬LITERAL( 'EXECUTE' ) THEN RETURN( '0'B);
    LINE=' EX#CUTE:' ;                             CALL OUT(LINE);
    IF ¬COMPOUND_STMT THEN CALL ERROR;
    IF #PROC THEN DO;
        LINE = ' END;' ;
        CALL OUT(LINE);
        #PROC = '0'B;
        END;
    ELSE DO;
        LINE = ' GO TO O#T;' ;
        CALL OUT(LINE);
        END;
    DO WHILE( LITERAL( ';' ) );
        IF ¬COMPOUND_STMT THEN CALL ERROR;
        IF #PROC THEN DO;
            LINE = ' END;' ;
            CALL OUT(LINE);
            #PROC = '0'B;
            END;
        ELSE DO;
            LINE = ' GO TO O#T;' ;
            CALL OUT(LINE);
        END;
    END;
END EXECUTE_BLOCK;

```

```

APPD1534
APPD1536
APPD1538
APPD1540
APPD1542
APPD1544
APPD1546
APPD1548
APPD1550
APPD1552
APPD1554
APPD1556
APPD1558
APPD1560
APPD1562
APPD1564
APPD1566
APPD1568
APPD1570
APPD1572
APPD1574
APPD1576
APPD1578
APPD1580
APPD1582
APPD1584
APPD1586
APPD1588
APPD1590
APPD1592
APPD1594
APPD1596
APPD1598
APPD1600
APPD1602
APPD1604
APPD1606
APPD1608

```


LINE = ' GO TO HE#D;' ;	CALL OUT(LINE);	APPD1686
RETURN('1'B);		APPD1688
END INITIALIZATION_STMT;		APPD1690
		APPD1692
MONITOR_STMT: PROC	RETURNS(BIT(1));	APPD1694
IF ~LITERAL('MONITOR') THEN RETURN('0'B);		APPD1696
LINE = ' O#T:' ;	CALL OUT(LINE);	APPD1698
LINE = ' PUT FILE(SYS\$PRINT) EDIT' ;	CALL OUT(LINE);	APPD1700
LINE = ' ('CYCLE = ',C#CLE)' ;	CALL OUT(LINE);	APPD1702
LINE = ' (SKIP(5),A,P''9999'');' ;	CALL OUT(LINE);	APPD1704
LINE = ' IF T#ME = 0 THEN' ;	CALL OUT(LINE);	APPD1706
LINE = ' PUT FILE(SYS\$PRINT) EDIT' ;	CALL OUT(LINE);	APPD1708
LINE = ' ('TIME = ',T#ME)' ;	CALL OUT(LINE);	APPD1710
LINE = ' (SKIP,A,P''9999999'');' ;	CALL OUT(LINE);	APPD1712
IF ~ID THEN CALL ERROR;		APPD1714
LINE = ' PUT FILE(SYS\$PRINT) DATA(' STAR ') SKIP;' ;		APPD1716
CALL OUT(LINE);		APPD1718
DO WHILE(LITERAL(', '));		APPD1720
IF ~ID THEN CALL ERROR;		APPD1722
LINE = ' PUT FILE(SYS\$PRINT) DATA(' STAR ') SKIP;' ;		APPD1724
CALL OUT(LINE);		APPD1726
END;		APPD1728
LINE = ' GO TO #NCR;' ;	CALL OUT(LINE);	APPD1730
RETURN('1'B);		APPD1732
END MONITOR_STMT;		APPD1734
		APPD1736
		APPD1738
DECLARATION_STMT: PROC	RETURNS(BIT(1));	APPD1740
IF REGISTER_DCL THEN GO TO L1;		APPD1742
IF SUBREGISTER_DCL THEN GO TO L1;		APPD1744
IF MEMORY_DCL THEN GO TO L1;		APPD1746
IF TERMINAL_DCL THEN GO TO L1;		APPD1748
IF CONSTANT_DCL THEN GO TO L1;		APPD1750
IF OPERATION_DCL THEN GO TO L1;		APPD1752
RETURN('0'B);		APPD1754
L1: RETURN('1'B);		APPD1756
END DECLARATION_STMT;		APPD1758
		APPD1760

REGISTER_DCL: PROC	RETURNS(BIT(1));	APPD1762
IF ~LITERAL('REGISTER') THEN RETURN('0'B);		APPD1764
IF ~REGISTER_ID THEN CALL ERROR;		APPD1766
DO WHILE(LITERAL(','));		APPD1768
IF ~REGISTER_ID THEN CALL ERROR;		APPD1770
END;		APPD1772
RETURN('1'B);		APPD1774
END REGISTER_DCL;		APPD1776
		APPD1778
REGISTER_ID: PROC	RETURNS(BIT(1));	APPD1780
DCL	(LENGTH,I1,I2) FIXED DECIMAL(2),	APPD1782
	TEMP CHAR(5),	APPD1784
	R CHAR(8) VARYING,	APPD1786
	L CHAR(2) VARYING;	APPD1788
IF ~ID THEN RETURN('0'B);		APPD1790
R = STAR;		APPD1792
IF ~LITERAL('(') THEN DO;		APPD1794
LINE = ' DCL ' R ' BIT(1);' ;		APPD1796
CALL OUT(LINE);		APPD1798
RETURN('1'B); END;		APPD1800
		APPD1802
IF ~INT THEN CALL ERROR;		APPD1804
I1 = STAR; /* CONVERSION PERFORMED HERE */		APPD1806
IF ~LITERAL('-') THEN CALL ERROR;		APPD1808
IF ~INT THEN CALL ERROR;		APPD1810
I2 = STAR; /* CONVERSION PERFORMED HERE */		APPD1812
IF ~LITERAL(')') THEN CALL ERROR;		APPD1814
LENGTH = I2 - I1 + 1;		APPD1816
TEMP = LENGTH;		APPD1818
IF LENGTH<=9 THEN L = SUBSTR(TEMP,5,1);		APPD1820
IF LENGTH >9 THEN L = SUBSTR(TEMP,4,2);		APPD1822
LINE = ' DCL ' R ' BIT(' L ');' ;		APPD1824
CALL OUT(LINE);		APPD1826
LENGTH = LENGTH - 1;		APPD1828
TEMP = LENGTH;		APPD1830
IF LENGTH<=9 THEN L = SUBSTR(TEMP,5,1);		APPD1832
IF LENGTH >9 THEN L = SUBSTR(TEMP,4,2);		APPD1834
LINE = ' DCL \$' R '(0:' L ') BIT(1) DEF ' R ';' ;		APPD1836

```

        CALL OUT(LINE);
        RETURN( '1'B );
END REGISTER_ID;

SUBREGISTER_DCL: PROC                RETURNS( BIT(1) );
    IF ~LITERAL('SUBREGISTER') THEN RETURN( '0'B );
    IF ~SREXP THEN CALL ERROR;
    DO WHILE( LITERAL(',') );
        IF ~SREXP THEN CALL ERROR;
    END;
    RETURN( '1'B );
END SUBREGISTER_DCL;

SREXP: PROC                            RETURNS( BIT(1) );
DCL          (R1,R2) CHAR(8) VARYING,
             BEGIN CHAR(2) VARYING,
             L     CHAR(2) VARYING,
             TEMP  CHAR(5),
             (LENGTH,I1,I2) FIXED DECIMAL(2);
    IF ~ID THEN RETURN( '0'B );
    IF ~LITERAL( '(' ) THEN CALL ERROR;
    IF ~ID THEN CALL ERROR;
    R2 = STAR;
    IF ~LITERAL( ')' ) THEN CALL ERROR;
    IF ~LITERAL( '=' ) THEN CALL ERROR;
    IF ~ID THEN CALL ERROR;
    R1 = STAR;
    IF ~LITERAL( '(' ) THEN CALL ERROR;
    IF ~INT THEN CALL ERROR;
    BEGIN = STAR;
    I1 = BEGIN; /* CONVERSION PERFORMED HERE */
    IF LITERAL( '"' ) THEN DO; L = '1'; GO TO L1; END;
    IF ~LITERAL( '-' ) THEN CALL ERROR;
    IF ~INT THEN CALL ERROR;
    I2 = STAR; /* CONVERSION PERFORMED HERE */
    IF ~LITERAL( ')' ) THEN CALL ERROR;
    LENGTH = I2 - I1 + 1;
    TEMP = LENGTH;

```

```

APPD1838
APPD1840
APPD1842
APPD1844
APPD1846
APPD1848
APPD1850
APPD1852
APPD1854
APPD1856
APPD1858
APPD1860
APPD1862
APPD1864
APPD1866
APPD1868
APPD1870
APPD1872
APPD1874
APPD1876
APPD1878
APPD1880
APPD1882
APPD1884
APPD1886
APPD1888
APPD1890
APPD1892
APPD1894
APPD1896
APPD1898
APPD1900
APPD1902
APPD1904
APPD1906
APPD1908
APPD1910
APPD1912

```

```

        IF LENGTH<=9 THEN L = SUBSTR(TEMP,5,1);
        IF LENGTH >9 THEN L = SUBSTR(TEMP,4,2);
L1: I1 = I1 + 1;
    TEMP = I1;
    IF I1<=9 THEN BEGIN = SUBSTR(TEMP,5,1);
    IF I1 >9 THEN BEGIN = SUBSTR(TEMP,4,2);
    LINE = ' DCL '||R2||' BIT('||L||') DEF '||R1||
           ' POSITION('||BEGIN||');' ;
    CALL OUT(LINE);
    /* INDIVIDUAL BITS OF A SUBREGISTER MAY NOT BE ACCESSED */
    RETURN( '1'B );
END SREXP;

MEMORY_DCL: PROC RETURNS( BIT(1) );
    IF ~LITERAL('MEMORY') THEN RETURN( '0'B );
    IF ~MEMEXP THEN CALL ERROR;
    DO WHILE( LITERAL(',' ) );
        IF ~MEMEXP THEN CALL ERROR;
    END;
    RETURN( '1'B );
END MEMORY_DCL;

MEMEXP: PROC RETURNS( BIT(1) );
    IF ~MEMLHS THEN RETURN( '0'B );
    IF ~LITERAL( '=' ) THEN CALL ERROR;
    IF ~MEMRHS THEN CALL ERROR;
    RETURN( '1'B );
END MEMEXP;

MEMLHS: PROC RETURNS( BIT(1) );
    IF ~ID THEN RETURN( '0'B );
    IF ~LITERAL( '(' ) THEN CALL ERROR;
    IF ~ID THEN CALL ERROR;
    IF LITERAL( ')' ) THEN RETURN( '1'B );
    IF ~LITERAL( '(' ) THEN CALL ERROR;
    IF ~ID THEN CALL ERROR;
    IF ~LITERAL( ')' ) THEN CALL ERROR;
    IF ~LITERAL( '(' ) THEN CALL ERROR;

```

```

APPD1914
APPD1916
APPD1918
APPD1920
APPD1922
APPD1924
APPD1926
APPD1928
APPD1930
APPD1932
APPD1934
APPD1936
APPD1938
APPD1940
APPD1942
APPD1944
APPD1946
APPD1948
APPD1950
APPD1952
APPD1954
APPD1956
APPD1958
APPD1960
APPD1962
APPD1964
APPD1966
APPD1968
APPD1970
APPD1972
APPD1974
APPD1976
APPD1978
APPD1980
APPD1982
APPD1984
APPD1986
APPD1988

```

RETURN('1'B);
END MEMLHS;

MEMRHS: PROC

DCL (I1,I2,I3,I4) RETURNS(BIT(1));
(LENGTH1,LENGTH2) FIXED DECIMAL(3),
R CHAR(8) VARYING,
TEMP CHAR(6),
(L1,L2) CHAR(3) VARYING;

IF ~ID THEN RETURN('0'B);
R = STAR;
IF ~LITERAL('(') THEN CALL ERROR;
IF ~INT THEN CALL ERROR;
I1 = STAR; /* CONVERSION PERFORMED HERE */
IF ~LITERAL('-') THEN CALL ERROR;
IF ~INT THEN CALL ERROR;
I2 = STAR; /* CONVERSION PERFORMED HERE */
LENGTH1 = I2 - I1 ;
TEMP = LENGTH1;
IF LENGTH1<=9 THEN L1 = SUBSTR(TEMP,6,1);
IF LENGTH1>99 THEN L1 = SUBSTR(TEMP,4,3);
IF LENGTH1 >9 & LENGTH1<=99 THEN L1 = SUBSTR(TEMP,5,2);
IF ~LITERAL(',') THEN CALL ERROR;
IF ~INT THEN CALL ERROR;
I3 = STAR; /* CONVERSION PERFORMED HERE */
IF ~LITERAL('-') THEN CALL ERROR;
IF ~INT THEN CALL ERROR;
I4 = STAR; /* CONVERSION PERFORMED HERE */
IF ~LITERAL(')') THEN CALL ERROR;
LENGTH2 = I4 - I3 + 1;
TEMP = LENGTH2;
IF LENGTH2<=9 THEN L2 = SUBSTR(TEMP,6,1);
IF LENGTH2>99 THEN L2 = SUBSTR(TEMP,4,3);
IF LENGTH2 >9 & LENGTH2<=99 THEN L2 = SUBSTR(TEMP,5,2);
LINE = ' DCL "||R||'(0: '||L1||') BIT('||L2||') ' ;
CALL OUT(LINE);
LENGTH1 = LENGTH1 + 1;
TEMP = LENGTH1;

APPD1990
APPD1992
APPD1994
APPD1996
APPD1998
APPD2000
APPD2002
APPD2004
APPD2006
APPD2008
APPD2010
APPD2012
APPD2014
APPD2016
APPD2018
APPD2020
APPD2022
APPD2024
APPD2026
APPD2028
APPD2030
APPD2032
APPD2034
APPD2036
APPD2038
APPD2040
APPD2042
APPD2044
APPD2046
APPD2048
APPD2050
APPD2052
APPD2054
APPD2056
APPD2058
APPD2060
APPD2062
APPD2064

```

LINE = ' INITIAL( ('||TEMP||') ('||L2||') '0'B );' ;
CALL OUT(LINE);
RETURN( '1'B );
END MEMRHS;

TERMINAL_DCL: PROC RETURNS( BIT(1) );
IF ~LITERAL('TERMINAL') THEN RETURN( '0'B );
IF ~REGISTER_ID THEN CALL ERROR;
DO WHILE( LITERAL( ', ' ) );
IF ~REGISTER_ID THEN CALL ERROR;
END;
RETURN( '1'B );
END TERMINAL_DCL;

CONSTANT_DCL: PROC RETURNS( BIT(1) );
IF ~LITERAL('CONSTANT') THEN RETURN( '0'B );
IF ~CONSEXP THEN CALL ERROR;
DO WHILE( LITERAL( ', ' ) );
IF ~CONSEXP THEN CALL ERROR;
END;
RETURN( '1'B );
END CONSTANT_DCL;

CONSEXP: PROC RETURNS( BIT(1) );
DCL R CHAR(8) VARYING;
IF ~ID THEN RETURN( '0'B );
R = STAR;
IF ~LITERAL( '=' ) THEN CALL ERROR;
IF ~INT THEN CALL ERROR;
LINE = ' DCL '||R||' FIX BIN(31) INITIAL('||STAR||');' ;
CALL OUT(LINE);
RETURN( '1'B );
END CONSEXP;

OPERATION_DCL: PROC RETURNS( BIT(1) );
IF ~LITERAL('OPERATION') THEN RETURN( '0'B );

```

```

APPD2066
APPD2068
APPD2070
APPD2072
APPD2074
APPD2076
APPD2078
APPD2080
APPD2082
APPD2084
APPD2086
APPD2088
APPD2090
APPD2092
APPD2094
APPD2096
APPD2098
APPD2100
APPD2102
APPD2104
APPD2106
APPD2108
APPD2110
APPD2112
APPD2114
APPD2116
APPD2118
APPD2120
APPD2122
APPD2124
APPD2126
APPD2128
APPD2130
APPD2132
APPD2134
APPD2136
APPD2138
APPD2140

```



```

/*      IF TERMINAL_STMT THEN GO TO L1;
      IF EXCHANGE_STMT THEN GO TO L1;
      RETURN( '0'B );
L1: IF ¬LITERAL('<') THEN GO TO L2;
      IF ¬INT THEN CALL ERROR;
      IF ¬LITERAL('>') THEN CALL ERROR;
      LINE = ' T#ME = MOD(T#ME + ' ||STAR|| ' , 1048576 );' ;
      CALL OUT(LINE);
L2: RETURN( '1'B );
END SIMPLE_STMT;

DO_STMT: PROC                RETURNS( BIT(1) ) RECURSIVE;
      IF ¬LITERAL('DO') THEN RETURN( '0'B );
      IF ¬ID THEN CALL ERROR;
      LINE = ' CALL ' ||STAR||';' ;
      CALL OUT(LINE);
      RETURN( '1'B );
END DO_STMT;

IF_STMT: PROC                RETURNS( BIT(1) ) RECURSIVE;
      IF ¬LITERAL('IF') THEN RETURN( '0'B );
      TEMP = ' IF ( ' ;
      IF ¬CONDITION THEN CALL ERROR;
      IF ¬LITERAL('THEN') THEN CALL ERROR;
      LINE = TEMP||' ) THEN ' ;
      CALL OUT(LINE);
      IF ¬SIMPLE_STMT THEN CALL ERROR;
      RETURN( '1'B );
END IF_STMT;

CONDITION: PROC                RETURNS( BIT(1) );
      IF ¬LITERAL( '(' ) THEN RETURN( '0'B );
      IF ¬BOOLEAN_TERM THEN CALL ERROR;
      IF ¬LITERAL( ')' ) THEN CALL ERROR;
      RETURN( '1'B );
END CONDITION;

BOOLEAN_TERM: PROC                RETURNS( BIT(1) ) RECURSIVE;

```

```

*/ APPD2218
APPD2220
APPD2222
APPD2224
APPD2226
APPD2228
APPD2230
APPD2232
APPD2234
APPD2236
APPD2238
APPD2240
APPD2242
APPD2244
APPD2246
APPD2248
APPD2250
APPD2252
APPD2254
APPD2256
APPD2258
APPD2260
APPD2262
APPD2264
APPD2266
APPD2268
APPD2270
APPD2272
APPD2274
APPD2276
APPD2278
APPD2280
APPD2282
APPD2284
APPD2286
APPD2288
APPD2290
APPD2292

```

```

    IF ¬BOOLEAN_FACTOR THEN RETURN( '0'B );
    DO WHILE( LITERAL( '.OR.' ) );
        TEMP = TEMP||' | ' ;
        IF ¬BOOLEAN_FACTOR THEN CALL ERROR;
    END;
    RETURN( '1'B );
END BOOLEAN_TERM;

BOOLEAN_FACTOR: PROC          RETURNS( BIT(1) ) RECURSIVE;
    IF ¬BOOLEAN_SEC THEN RETURN( '0'B );
    DO WHILE( LITERAL( '.AND.' ) );
        TEMP = TEMP||' & ' ;
        IF ¬BOOLEAN_SEC THEN CALL ERROR;
    END;
    RETURN( '1'B );
END BOOLEAN_FACTOR;

BOOLEAN_SEC: PROC            RETURNS( BIT(1) ) RECURSIVE;
    IF LITERAL( '.NOT.' ) THEN TEMP = TEMP||' ¬ ' ;
    IF BOOLEAN_PRIM THEN GO TO L1;
    RETURN( '0'B );
    L1: RETURN( '1'B );
END BOOLEAN_SEC;

BOOLEAN_PRIM: PROC          RETURNS( BIT(1) ) RECURSIVE;
    IF LOGICAL_VALUE THEN RETURN( '1'B );
    IF RELATION THEN RETURN( '1'B );
    IF ¬LITERAL( '(' ) THEN RETURN( '0'B );
    TEMP = TEMP||' ( ' ;
    IF ¬BOOLEAN_TERM THEN CALL ERROR;
    IF ¬LITERAL( ')' ) THEN CALL ERROR;
    TEMP = TEMP||' ) ' ;
    RETURN( '1'B );
END BOOLEAN_PRIM;

RELATION: PROC              RETURNS( BIT(1) ) RECURSIVE;
    IF ¬SAE THEN RETURN( '0'B );
    TEMP = TEMP||SO;

```

```

APPD2294
APPD2296
APPD2298
APPD2300
APPD2302
APPD2304
APPD2306
APPD2308
APPD2310
APPD2312
APPD2314
APPD2316
APPD2318
APPD2320
APPD2322
APPD2324
APPD2326
APPD2328
APPD2330
APPD2332
APPD2334
APPD2336
APPD2338
APPD2340
APPD2342
APPD2344
APPD2346
APPD2348
APPD2350
APPD2352
APPD2354
APPD2356
APPD2358
APPD2360
APPD2362
APPD2364
APPD2366
APPD2368

```

```

        IF ¬REL_OP THEN RETURN( '1'B );
        IF ¬SAE THEN CALL ERROR;
        TEMP = TEMP||SO;
        RETURN( '1'B );
END RELATION;

REL_OP: PROC                                RETURNS( BIT(1) )    RECURSIVE;
    IF LITERAL( '.EQ.' ) THEN DO;
        TEMP = TEMP||' = ' ; GO TO L1; END;
    IF LITERAL( '.NE.' ) THEN DO;
        TEMP = TEMP||' ¬= ' ; GO TO L1; END;
    IF LITERAL( '.LT.' ) THEN DO;
        TEMP = TEMP||' < ' ; GO TO L1; END;
    IF LITERAL( '.GT.' ) THEN DO;
        TEMP = TEMP||' > ' ; GO TO L1; END;
    IF LITERAL( '.LE.' ) THEN DO;
        TEMP = TEMP||' ≤ ' ; GO TO L1; END;
    IF LITERAL( '.GE.' ) THEN DO;
        TEMP = TEMP||' ≥ ' ; GO TO L1; END;
    RETURN( '0'B );
L1: RETURN( '1'B );
END REL_OP;

SAE: PROC                                    RETURNS( BIT(1) )    RECURSIVE;
    SO = " ";
    IF INT THEN DO; SO = STAR; GO TO L1; END;
    IF DEST THEN GO TO L1;
    RETURN( '0'B );
L1: RETURN( '1'B );
END SAE;

LOGICAL_VALUE: PROC                          RETURNS( BIT(1) );
    IF LITERAL( '.TRUE.' ) THEN DO;
        TEMP = TEMP||' '1'B ' ; GO TO L1; END;
    IF LITERAL( '.FALSE.' ) THEN DO;
        TEMP = TEMP||' '0'B ' ; GO TO L1; END;
    RETURN( '0'B );
L1: RETURN( '1'B );

```

```

APPD2370
APPD2372
APPD2374
APPD2376
APPD2378
APPD2380
APPD2382
APPD2384
APPD2386
APPD2388
APPD2390
APPD2392
APPD2394
APPD2396
APPD2398
APPD2400
APPD2402
APPD2404
APPD2406
APPD2408
APPD2410
APPD2412
APPD2414
APPD2416
APPD2418
APPD2420
APPD2422
APPD2424
APPD2426
APPD2428
APPD2430
APPD2432
APPD2434
APPD2436
APPD2438
APPD2440
APPD2442
APPD2444

```

```

END LOGICAL_VALUE;

GO_TO_STMT: PROC RETURNS( BIT(1) ) RECURSIVE;
  IF ~LITERAL( 'GO' ) THEN RETURN( '0'B );
  IF ~LITERAL( 'TO' ) THEN CALL ERROR;
  IF ~ID THEN CALL ERROR;
  LINE = ' GO TO ' || STAR || '; ' ; CALL OUT(LINE);
  RETURN( '1'B );
END GO_TO_STMT;

TRANSFER_STMT: PROC RETURNS( BIT(1) );
  DCL BACKUP FIXED BINARY(31);
  BACKUP = I; /* SAVE PTR TO TEXT POSITION */
  TYPE = -1; SO = ' '; S1 = ' '; S2 = ' ';

  IF ~DEST THEN RETURN( '0'B );
  IF ~LITERAL( '<' ) THEN DO;
    I = BACKUP; /* BACKUP TEXT PTR */
    RETURN( '0'B ); END;
  IF INT THEN DO;
    L1: LINE = ' CALL #SET(' || SO || ', ' || STAR || '); ' ;
    GO TO DONE;
    END;

  IF UNARY_EXP THEN DO;
    IF TYPE = 2 THEN GO TO L2;
    IF TYPE = 3 THEN GO TO L3;
    IF TYPE = 4 THEN GO TO L4;
    IF TYPE = 5 THEN GO TO L5;
    IF TYPE = 6 THEN GO TO L6;
    IF TYPE = 7 THEN GO TO L7;
    IF TYPE = 8 THEN GO TO L8;
    CALL ERROR;
    END;

  IF BINARY_EXP THEN DO;
    IF TYPE = 0 THEN GO TO L0;
    IF TYPE = 59 THEN GO TO L9;

```

```

APPD2446
APPD2448
APPD2450
APPD2452
APPD2454
APPD2456
APPD2458
APPD2460
APPD2462
APPD2464
APPD2466
APPD2468
APPD2470
APPD2472
APPD2474
APPD2476
APPD2478
APPD2480
APPD2482
APPD2484
APPD2486
APPD2488
APPD2490
APPD2492
APPD2494
APPD2496
APPD2498
APPD2500
APPD2502
APPD2504
APPD2506
APPD2508
APPD2510
APPD2512
APPD2514
APPD2516
APPD2518
APPD2520

```

```

IF TYPE = 60 THEN GO TO L10;
IF TYPE = 61 THEN GO TO L11;
IF TYPE = 62 THEN GO TO L12;
IF TYPE = 63 THEN GO TO L13;
IF TYPE = 64 THEN GO TO L14;
IF TYPE = 65 THEN GO TO L15;
IF TYPE = 66 THEN GO TO L16;
CALL ERROR;

```

END;

CALL ERROR;

```

L0: LINE = ' ||S0||' = ' ||S1||';' ;
GO TO DONE;
L2: LINE = ' CALL UA#D(' ||S0||', ' ||S2||');' ;
GO TO DONE;
L3: LINE = ' CALL #UOR(' ||S0||', ' ||S2||');' ;
GO TO DONE;
L4: LINE = ' ||S0||' = ' ||S2||';' ;
GO TO DONE;
L5: LINE = ' CALL US#L(' ||S0||', ' ||S2||');' ;
GO TO DONE;
L6: LINE = ' CALL US#R(' ||S0||', ' ||S2||');' ;
GO TO DONE;
L7: LINE = ' CALL CI#L(' ||S0||', ' ||S2||');' ;
GO TO DONE;
L8: LINE = ' CALL CI#R(' ||S0||', ' ||S2||');' ;
GO TO DONE;

L9: LINE = ' CALL #ADD(' ||S0||', ' ||S1||', ' ||S2||');' ;
GO TO DONE;
L10: LINE = ' CALL #SUB(' ||S0||', ' ||S1||', ' ||S2||');' ;
GO TO DONE;
L11: LINE = ' CALL #BOR(' ||S0||', ' ||S1||', ' ||S2||');' ;
GO TO DONE;
L12: LINE = ' CALL BA#D(' ||S0||', ' ||S1||', ' ||S2||');' ;
GO TO DONE;
L13: LINE = ' CALL BX#R(' ||S0||', ' ||S1||', ' ||S2||');' ;

```

```

APPD2522
APPD2524
APPD2526
APPD2528
APPD2530
APPD2532
APPD2534
APPD2536
APPD2538
APPD2540
APPD2542
APPD2544
APPD2546
APPD2548
APPD2550
APPD2552
APPD2554
APPD2556
APPD2558
APPD2560
APPD2562
APPD2564
APPD2566
APPD2568
APPD2570
APPD2572
APPD2574
APPD2576
APPD2578
APPD2580
APPD2582
APPD2584
APPD2586
APPD2588
APPD2590
APPD2592
APPD2594
APPD2596

```

GO TO DONE;	APPD2598
L14: LINE = ' CALL BS#L(' SO ',' S1 ',' S2 ');' ;	APPD2600
GO TO DONE;	APPD2602
L15: LINE = ' CALL BS#R(' SO ',' S1 ',' S2 ');' ;	APPD2604
GO TO DONE;	APPD2606
L16: LINE = ' CALL #CNT(' SO ',' S1 ',' S2 ');' ;	APPD2608
GO TO DONE;	APPD2610
	APPD2612
DONE: CALL OUT(LINE);	APPD2614
RETURN('1'B);	APPD2616
END TRANSFER_STMT;	APPD2618
	APPD2620
DEST: PROC RETURNS(BIT(1));	APPD2622
DCL R CHAR(8) VARYING;	APPD2624
DCL I1 CHAR(8) VARYING;	APPD2626
IF ~ID THEN RETURN('0'B);	APPD2628
R = STAR;	APPD2630
IF ~LITERAL('(') THEN DO; SO = R;	APPD2632
GO TO L1;	APPD2634
END;	APPD2636
IF ID THEN DO;	APPD2638
IF LITERAL(')') THEN DO;	APPD2640
SO = R '(' STAR ')' ;	APPD2642
GO TO L1;	APPD2644
ELSE CALL ERROR;	APPD2646
END;	APPD2648
IF ~INT THEN CALL ERROR;	APPD2650
I1 = STAR;	APPD2652
IF LITERAL('\$') THEN DO; SO='\$' R '(' STAR ')';	APPD2654
GO TO L1;	APPD2656
END;	APPD2658
IF ~LITERAL('-') THEN CALL ERROR;	APPD2660
IF ~INT THEN CALL ERROR;	APPD2662
IF ~LITERAL(')') THEN CALL ERROR;	APPD2664
SO = "SUBSTR(' R ',' I1 '+1,' STAR '-' I1 '+1)' ;	APPD2666
GO TO L1;	APPD2668
L1: RETURN('1'B);	APPD2670
END DEST;	APPD2672

UNARY_EXP: PROC	RETURNS(BIT(1));	APPD2674
IF ~UNARY_OP THEN RETURN('0'B);		APPD2676
IF ~UNARY_SOURCE THEN CALL ERROR;		APPD2678
RETURN('1'B);		APPD2680
END UNARY_EXP;		APPD2682
		APPD2684
		APPD2686
		APPD2688
UNARY_OP: PROC	RETURNS(BIT(1));	APPD2690
IF LITERAL(' .AND.') THEN DO; TYPE = 2; GO TO L1; END;		APPD2692
IF LITERAL(' .OR.') THEN DO; TYPE = 3; GO TO L1; END;		APPD2694
IF LITERAL(' .NOT.') THEN DO; TYPE = 4; GO TO L1; END;		APPD2696
IF LITERAL(' .SHL.') THEN DO; TYPE = 5; GO TO L1; END;		APPD2698
IF LITERAL(' .SHR.') THEN DO; TYPE = 6; GO TO L1; END;		APPD2700
IF LITERAL(' .CIRL.') THEN DO; TYPE = 7; GO TO L1; END;		APPD2702
IF LITERAL(' .CIRR.') THEN DO; TYPE = 8; GO TO L1; END;		APPD2704
RETURN('0'B);		APPD2706
L1: RETURN('1'B);		APPD2708
END UNARY_OP;		APPD2710
		APPD2712
		APPD2714
		APPD2716
UNARY_SOURCE: PROC	RETURNS(BIT(1));	APPD2718
DCL R CHAR(8) VARYING;		APPD2720
DCL I1 CHAR(8) VARYING;		APPD2722
IF ~ID THEN RETURN('0'B);		APPD2724
R = STAR;		APPD2726
IF ~LITERAL('(') THEN DO; S2 = R;		APPD2728
GO TO L1;		APPD2730
END;		APPD2732
IF ID THEN DO;		APPD2734
IF LITERAL('(') THEN DO;		APPD2736
S2 = R '(' STAR ')';		APPD2738
GO TO L1; END;		APPD2740
ELSE CALL ERROR;		APPD2742
END;		APPD2744
IF ~INT THEN CALL ERROR;		APPD2746
I1 = STAR;		APPD2748
IF LITERAL('(') THEN DO; S2 = '\$' R '(' STAR ')';		
GO TO L1;		
END;		

```

        IF ~LITERAL('-') THEN CALL ERROR;
        IF ~INT THEN CALL ERROR;
        IF ~LITERAL(')') THEN CALL ERROR;
        S2 = 'SUBSTR('||R||','||I1||'+1,'||STAR||'-'||I1||'+1)';
        GO TO L1;
    L1: RETURN( '1'B );
END UNARY_SOURCE;

BINARY_EXP: PROC                                RETURNS( BIT(1) );
    IF ~OP1 THEN RETURN( '0'B );
    IF ~BINARY_OP THEN DO; TYPE = 0; GO TO L1; END;
    IF OP2 THEN GO TO L1;
    IF INT THEN DO; S2 = STAR; GO TO L1; END;
    CALL ERROR;
    L1: RETURN( '1'B );
END BINARY_EXP;

BINARY_OP: PROC                                RETURNS( BIT(1) );
    IF LITERAL('.ADD.') THEN DO; TYPE = 59; GO TO L1; END;
    IF LITERAL('.SUB.') THEN DO; TYPE = 60; GO TO L1; END;
    IF LITERAL('.OR.') THEN DO; TYPE = 61; GO TO L1; END;
    IF LITERAL('.AND.') THEN DO; TYPE = 62; GO TO L1; END;
    IF LITERAL('.XOR.') THEN DO; TYPE = 63; GO TO L1; END;
    IF LITERAL('.SHL.') THEN DO; TYPE = 64; GO TO L1; END;
    IF LITERAL('.SHR.') THEN DO; TYPE = 65; GO TO L1; END;
    IF LITERAL('.CNT.') THEN DO; TYPE = 66; GO TO L1; END;
    RETURN( '0'B );
    L1: RETURN( '1'B );
END BINARY_OP;

OP1: PROC                                       RETURNS( BIT(1) );
    DCL      R          CHAR(8)  VARYING;
    DCL      I1         CHAR(8)  VARYING;
    IF ~ID THEN RETURN( '0'B );
    R = STAR;
    IF ~LITERAL( '(' ) THEN DO; S1 = R;
                                GO TO L1;
                                END;

```

```

APPD2750
APPD2752
APPD2754
APPD2756
APPD2758
APPD2760
APPD2762
APPD2764
APPD2766
APPD2768
APPD2770
APPD2772
APPD2774
APPD2776
APPD2778
APPD2780
APPD2782
APPD2784
APPD2786
APPD2788
APPD2790
APPD2792
APPD2794
APPD2796
APPD2798
APPD2800
APPD2802
APPD2804
APPD2806
APPD2808
APPD2810
APPD2812
APPD2814
APPD2816
APPD2818
APPD2820
APPD2822
APPD2824

```

```

IF ID THEN DO;
    IF LITERAL( '(' ) THEN DO;
        S1 = R( '(' || STAR || ' ' );
        GO TO L1;
    ELSE CALL ERROR;
    END;
IF ~INT THEN CALL ERROR;
I1 = STAR;
IF LITERAL( '(' ) THEN DO; S1 = '$' || R( '(' || STAR || ' ' );
    GO TO L1;
    END;
IF ~LITERAL( '-' ) THEN CALL ERROR;
IF ~INT THEN CALL ERROR;
IF ~LITERAL( '(' ) THEN CALL ERROR;
S1 = "SUBSTR( ' || R( '(' || I1 || '+1, ' || STAR || '- ' || I1 || '+1 ) ' ";
GO TO L1;
L1: RETURN( '1'B );
END OP1;

OP2: PROC
    RETURNS( BIT(1) );
DCL R CHAR(8) VARYING;
DCL I1 CHAR(8) VARYING;
IF ~ID THEN RETURN( '0'B );
R = STAR;
IF ~LITERAL( '(' ) THEN DO; S2 = R;
    GO TO L1;
    END;
IF ID THEN DO;
    IF LITERAL( '(' ) THEN DO;
        S2 = R( '(' || STAR || ' ' );
        GO TO L1;
    ELSE CALL ERROR;
    END;
IF ~INT THEN CALL ERROR;
I1 = STAR;
IF LITERAL( '(' ) THEN DO; S2 = '$' || R( '(' || STAR || ' ' );
    GO TO L1;
    END;
END;

```

```

APPD2826
APPD2828
APPD2830
APPD2832
APPD2834
APPD2836
APPD2838
APPD2840
APPD2842
APPD2844
APPD2846
APPD2848
APPD2850
APPD2852
APPD2854
APPD2856
APPD2858
APPD2860
APPD2862
APPD2864
APPD2866
APPD2868
APPD2870
APPD2872
APPD2874
APPD2876
APPD2878
APPD2880
APPD2882
APPD2884
APPD2886
APPD2888
APPD2890
APPD2892
APPD2894
APPD2896
APPD2898
APPD2900

```

IF ¬LITERAL('¬') THEN CALL ERROR;	APPD2902
IF ¬INT THEN CALL ERROR;	APPD2904
IF ¬LITERAL(')')	APPD2906
S2 = 'SUBSTR(' R ',' I '+1,' STAR '-' I '+1)';	APPD2908
GO TO L1;	APPD2910
L1: RETURN('1'B);	APPD2912
END OP2;	APPD2914
TERMINAL_STMT: PROC RETURNS(BIT(1));	APPD2916
DCL BACKUP FIXED BINARY(31);	APPD2918
SO = ' '; S2 = ' ';	APPD2920
BACKUP = I; /* SAVE PTR TO TEXT POSITION */	APPD2922
IF ¬TERMINAL THEN RETURN('0'B);	APPD2924
IF ¬LITERAL(':=') THEN DO;	APPD2926
I = BACKUP; /* BACKUP TEXT PTR */	APPD2928
RETURN('0'B); END;	APPD2930
TEMP = ' ' SO ' = ' ;	APPD2932
IF ¬LABEL THEN CALL ERROR;	APPD2934
LINE = TEMP ';' ;	APPD2936
CALL OUT(LINE);	APPD2938
RETURN('1'B);	APPD2940
END TERMINAL_STMT;	APPD2942
TERMINAL: PROC RETURNS(BIT(1));	APPD2944
DCL R CHAR(8) VARYING;	APPD2946
IF ¬ID THEN RETURN('0'B);	APPD2948
R = STAR;	APPD2950
IF ¬LITERAL('(') THEN DO; SO = R; GO TO L1; END;	APPD2952
IF ¬INT THEN CALL ERROR;	APPD2954
IF ¬LITERAL(')') THEN CALL ERROR;	APPD2956
SO = '\$' R '(' STAR)';	APPD2958
L1: RETURN('1'B);	APPD2960
END TERMINAL;	APPD2962
LABEL: PROC RETURNS(BIT(1));	APPD2964
IF ¬TERM THEN RETURN('0'B);	APPD2966
L: IF ¬BOP THEN GO TO L1;	APPD2968
IF ¬TERM THEN CALL ERROR;	APPD2970
	APPD2972
	APPD2974
	APPD2976

```

        GO TO L1;
    L1: RETURN( '1'B );
END LABEL;

TERM:  PROC                                RETURNS( BIT(1) );
        IF DEF1 THEN GO TO L1;
        IF DEF2 THEN GO TO L1;
        RETURN( '0'B );
    L1: RETURN( '1'B );
END TERM;

DEF1:  PROC                                RETURNS( BIT(1) );
        DCL      R          CHAR(8) VARYING;
        IF ~ID THEN RETURN( '0'B );
        R = STAR;
        IF ~LITERAL( '(' ) THEN DO; S1 = R; GO TO L1; END;
        IF ~INT THEN CALL ERROR;
        IF ~LITERAL( ')' ) THEN CALL ERROR;
        S1 = '$' || R || '(' || STAR || ')';
    L1: TEMP = TEMP || S1;
        RETURN( '1'B );
END DEF1;

DEF2:  PROC                                RETURNS( BIT(1) );
        IF ~LITERAL( '(' ) THEN RETURN( '0'B );
        IF ~LITERAL( '~' ) THEN CALL ERROR;
        TEMP = TEMP || '~';
        IF ~DEF1 THEN CALL ERROR;
        IF ~LITERAL( ')' ) THEN CALL ERROR;
        RETURN( '1'B );
END DEF2;

BOP:  PROC                                RETURNS( BIT(1) );
        IF LITERAL( '*' ) THEN DO; TEMP=TEMP || '&'; GO TO L1; END;
        IF LITERAL( '+' ) THEN DO; TEMP=TEMP || '|'; GO TO L1; END;
        RETURN( '0'B );
    L1: RETURN( '1'B );
END BOP;

```

```

APPD2978
APPD2980
APPD2982
APPD2984
APPD2986
APPD2988
APPD2990
APPD2992
APPD2994
APPD2996
APPD2998
APPD3000
APPD3002
APPD3004
APPD3006
APPD3008
APPD3010
APPD3012
APPD3014
APPD3016
APPD3018
APPD3020
APPD3022
APPD3024
APPD3026
APPD3028
APPD3030
APPD3032
APPD3034
APPD3036
APPD3038
APPD3040
APPD3042
APPD3044
APPD3046
APPD3048
APPD3050
APPD3052

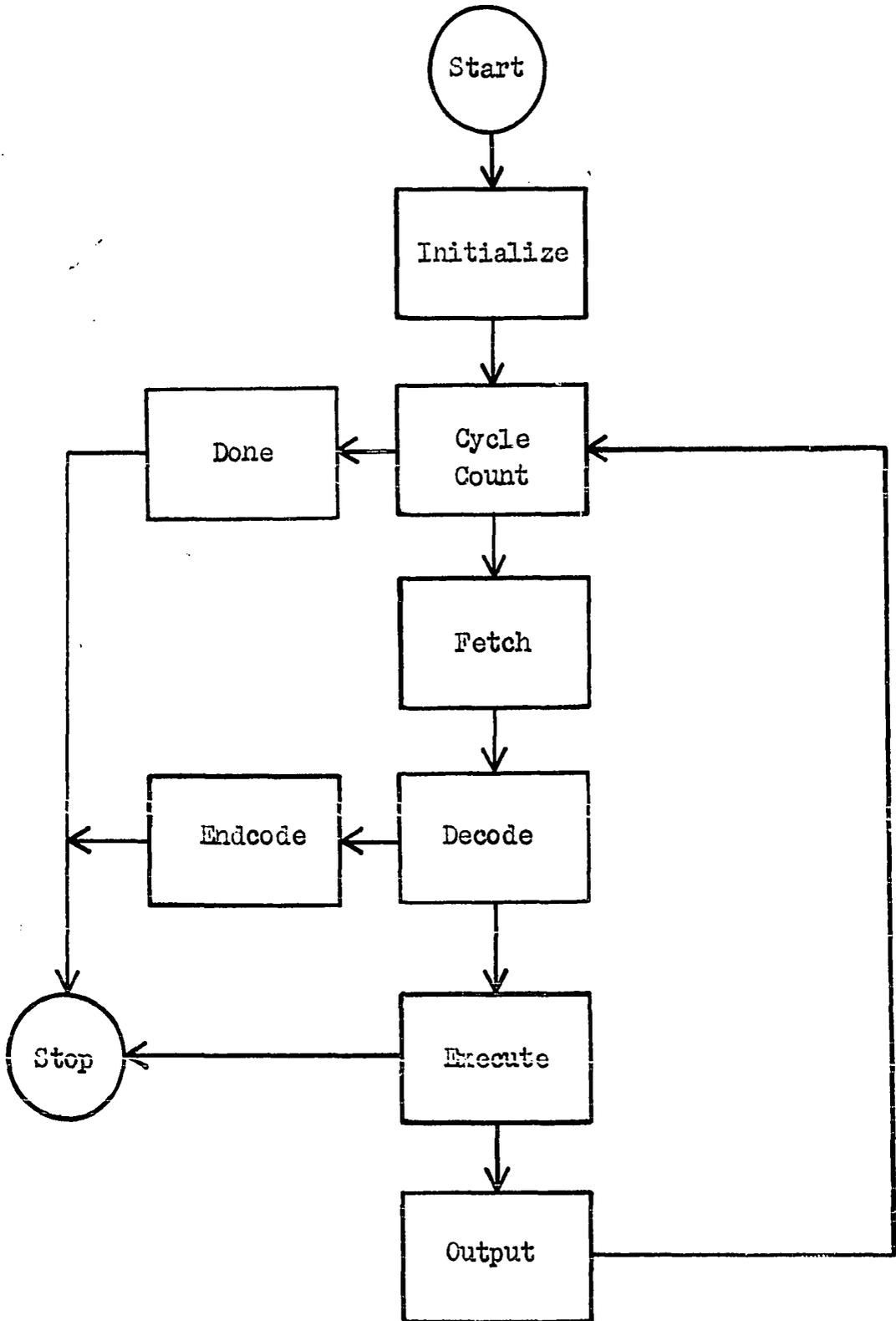
```

/* THIS IS THE END OF THE INTERNAL PROCEDURES */

FINIS: PUT EDIT('END OF FILE SENSED ON SYSIN')(SKIP(2),COL(10),A);
DONE:
END COMPILE;

APPD3054
APPD3056
APPD3058
APPD3060
APPD3062
APPD3064
APPD3066
APPD3068
APPD3070
APPD3072
APPD3074

Appendix 5. Simulator Flow Chart



Appendix 6. DCLS

GO TO #NCR;
/* END OF STANDARD DECLARES */

APPF1390
APPF1400

Appendix 7. ENDCODE

Appendix 8. PRIMPROC


```

        (SKIP(5),COL(10),A);
DO I = 1 TO LENGTH(SOURCE) BY 1;
    IF SUBSTR(SOURCE,I,1) = '1'B THEN GO TO L1;
    END;
    DEST = '0'B;
    RETURN;
L1: DEST = '1'B;
    END #UOR;

US#L: PROC(DEST,SOURCE);
    DCL (DEST,SOURCE) BIT(*);
    IF LENGTH(DEST) /= LENGTH(SOURCE) THEN
        PUT FILE(SYSPRINT) EDIT
        ('*** LENGTH ERROR -- US#L PROCEDURE ***')
        (SKIP(5),COL(10),A);
    SUBSTR(DEST,1,LENGTH(DEST)-1) = SUBSTR(SOURCE,2,LENGTH(SOURCE)-1);
    SUBSTR(DEST,LENGTH(DEST),1) = '0'B; /* ZERO FILL */
    END US#L;

US#R: PROC(DEST,SOURCE);
    DCL (DEST,SOURCE) BIT(*);
    IF LENGTH(DEST) /= LENGTH(SOURCE) THEN
        PUT FILE(SYSPRINT) EDIT
        ('*** LENGTH ERROR -- US#R PROCEDURE ***')
        (SKIP(5),COL(10),A);
    SUBSTR(DEST,2) = SUBSTR(SOURCE,1,LENGTH(SOURCE)-1);
    SUBSTR(DEST,1,1) = '0'B; /* ZERO FILL */
    END US#R;

CI#L: PROC(DEST,SOURCE);
    DCL (DEST,SOURCE) BIT(*),
        TEMP BIT(1);
    IF LENGTH(DEST) /= LENGTH(SOURCE) THEN
        PUT FILE(SYSPRINT) EDIT
        ('*** LENGTH ERROR -- CI#L PROCEDURE ***')

```

```

APPH139C
APPH140C
APPH1410
APPH1420
APPH1430
APPH1440
APPH1450
APPH1460
APPH1470
APPH1480
APPH1490
APPH1500
APPH1510
APPH1520
APPH1530
APPH1540
APPH1550
APPH1560
APPH1570
APPH1580
APPH1590
APPH1600
APPH1610
APPH1620
APPH1630
APPH1640
APPH1650
APPH1660
APPH1670
APPH1680
APPH1690
APPH1700
APPH1710
APPH1720
APPH1730
APPH1740
APPH1750
APPH1760

```

```

        (SKIP(5),COL(10),A);
        TEMP = SUBSTR(SOURCE,1,1);
SUBSTR(DEST,1,LENGTH(DEST)-1) = SUBSTR(SOURCE,2,LENGTH(SOURCE)-1);
        SUBSTR(DEST,LENGTH(DEST),1) = TEMP;      /* END AROUND */
        END CI#L;

CI#R: PROC(DEST,SOURCE);
        DCL      (DEST,SOURCE)      BIT(*),
                TEMP      BIT(1);
        IF LENGTH(DEST) /= LENGTH(SOURCE) THEN
                PUT FILE(SYSPRINT) EDIT
                ( '*** LENGTH ERROR -- CI#R PROCEDURE ***' )
                (SKIP(5),COL(10),A);
        TEMP = SUBSTR(SOURCE,LENGTH(SOURCE),1);
SUBSTR(DEST,2,LENGTH(DEST)-1) = SUBSTR(SOURCE,1,LENGTH(SOURCE)-1);
        SUBSTR(DEST,1,1) = TEMP;      /* END AROUND */
        END CI#R;

#ADD: PROC(DEST,S1,S2);
        DCL      (DEST,S1,S2)      BIT(*);
        #TEMP = S1 + S2;      /* CONVERSION HERE */
        DEST = SUBSTR( #TEMP, 32 - LENGTH(DEST) );
        OVERFLOW = SUBSTR( #TEMP, 31 - LENGTH(DEST), 1 );
        END #ADD;

#SUB: PROC(DEST,OP1,OP2);
        DCL      (DEST,OP1,OP2)      BIT(*);
        IF OP2 > OP1 THEN
                PUT FILE(SYSPRINT) EDIT
                ( '*** MAGNITUDE ERROR -- #SUB PROCEDURE ***' )
                (SKIP(5),COL(10),A);
        #TEMP = OP1 - OP2;      /* CONVERSION HERE */
        DEST = SUBSTR(#TEMP, 32-LENGTH(DEST));
        END #SUB;

```

```

APFH177C
APPH1780
APPH1790
APPH180C
APPH1810
APPH1820
APPH1830
APPH1840
APPH1850
APPH1860
APPH1870
APPH1880
APPH189C
APPH1900
APPH191C
APPH1920
APPH1930
APPH1940
APPH195C
APPH1960
APPH1970
APPH1980
APPH1990
APPH200C
APPH2010
APPH2020
APPH2030
APPH2040
APPH205C
APPH2060
APPH2070
APPH2080
APPH2090
APPH210C
APPH2110
APPH212C
APPH2130
APPH2140

```

#BOR: PROC(DEST,OP1,OP2);	APPH2150
DCL (DEST,OP1,OP2) BIT(*);	APPH2160
IF LENGTH(OP1) /= LENGTH(OP2) THEN	APPH2170
PUT FILE(SYSPRINT) EDIT	APPH2180
('*** LENGTH ERROR -- #BOR PROCEDURE ***')	APPH2190
(SKIP(5),COL(10),A);	APPH2200
DEST = OP1 OP2;	APPH2210
END #BOR;	APPH2220
	APPH2230
	APPH2240
	APPH2250
BA#D: PROC(DEST,OP1,OP2);	APPH2260
DCL (DEST,OP1,OP2) BIT(*);	APPH2270
IF LENGTH(OP1) /= LENGTH(OP2) THEN	APPH2280
PUT FILE(SYSPRINT) EDIT	APPH2290
('*** LENGTH ERROR -- BA#D PROCEDURE ***')	APPH2300
(SKIP(5),COL(10),A);	APPH2310
DEST = OP1 & OP2;	APPH2320
END BA#D;	APPH2330
	APPH2340
	APPH2350
BX#R: PROC(DEST,OP1,OP2);	APPH2360
DCL (DEST,OP1,OP2) BIT(*);	APPH2370
IF LENGTH(OP1) /= LENGTH(OP2) THEN	APPH2380
PUT FILE(SYSPRINT) EDIT	APPH2390
('*** LENGTH ERROR -- BX#R PROCEDURE ***')	APPH2400
(SKIP(5),COL(10),A);	APPH2410
DO I = 1 TO LENGTH(OP1) BY 1;	APPH2420
IF SUBSTR(OP1,I,1)='1'B & SUBSTR(OP2,I,1)='1'B	APPH2430
THEN SUBSTR(DEST,I,1) = '1'B;	APPH2440
ELSE SUBSTR(DEST,I,1) = '0'B;	APPH2450
END;	APPH2460
END BX#R;	APPH2470
	APPH2480
	APPH2490
BS#L: PROC(DEST,SOURCE,INT);	APPH2500
DCL (DEST,SOURCE) BIT(*),	APPH2510
INT FIXED BINARY(31);	APPH2520

IF LENGTH(DEST) /= LENGTH(SOURCE) THEN	APPH2530
PUT FILE(SYSPRINT) EDIT	APPH2540
('*** LENGTH ERROR -- BS#L PROCEDURE ***')	APPH2550
(SKIP(5),COL(10),A);	APPH2560
IF INT>=LENGTH(SOURCE) THEN DO; DEST='0'B; GO TO L1; END;	APPH2570
DEST = SUBSTR(SOURCE,INT+1);	APPH2580
L1: END BS#L;	APPH2590
	APPH2600
	APPH2610
BS#R: PROC(DEST,SOURCE,INT);	APPH2620
DCL (DEST,SOURCE) BIT(*),	APPH2630
INT FIXED BINARY(31);	APPH2640
IF LENGTH(DEST) /= LENGTH(SOURCE) THEN	APPH2650
PUT FILE(SYSPRINT) EDIT	APPH2660
('*** LENGTH ERROR -- BS#R PROCEDURE ***')	APPH2670
(SKIP(5),COL(10),A);	APPH2680
IF INT>=LENGTH(SOURCE) THEN DO; DEST='0'B; GO TO L1; END;	APPH2690
#TEMP = SOURCE/(2**INT); /* CONVERSION HERE */	APPH2700
DEST = SUBSTR(#TEMP, 32 - LENGTH(DEST));	APPH2710
L1: END BS#R;	APPH2720
	APPH2730
	APPH2740
	APPH2750
/* THIS IS THE END OF THE PRIMITIVE OPERATIONS */	