Report No. 469

IBAL MANUAL*

The Illiac III Basic Assembler Language

by

John C. Schwebel

revised by

Ahmad E. Masumi

July, 1971

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

*Supported in part by Contract AT(11-1)-2118 with the United States Atomic Energy Commission.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

IBAL MANUAL

Table of Contents

1. Introduction
2. Definition of the Language
 - 2.1 Terminal Symbols
 - 2.2 Basic Elements
 - 2.3 Program Structure
 - 2.4 Declarations
 - 2.5 Instructions
 - 2.6 Commands
 - 2.7 Directives
3. Segmentation
 - 3.1 Procedure Segments
 - 3.2 Data Segments
 - 3.3 Storage Attributes
 - 3.4 Intersegment Communication
4. Pointer Register Allocation

- Appendix 1 - The 60 Character Set
- Appendix 2 - Reserved Words
- Appendix 3 - Complete IBAL Syntax
- Appendix 4 - Sample Program Segment
- Appendix 5 - Instruction Mnemonic Code

Date: 8/4/71
Section: Table of Contents
Page 1 of 1
Revision:
IBAL MANUAL

2. Definition of the Language

Contents

- 2.1 Terminal Symbols
 - 2.1.1 Basic Characters
 - 2.1.2 Compound Punctuations
 - 2.1.3 Reserved Words
- 2.2 Basic Elements
 - 2.2.1 Comments and Blanks
 - 2.2.2 Identifiers
 - 2.2.3 Unsigned Integers
 - 2.2.4 Constants
 - 2.2.5 Literals
 - 2.2.6 Self-Defining Values
 - 2.2.7 Expressions
- 2.3 Program Strucutre
 - 2.3.1 Blocks
 - 2.3.2 Procedures
 - 2.3.3 Scope
 - 2.3.4 Storage Allocation
- 2.4 Declarations
 - 2.4.1 Attributes
 - 2.4.2 Conditional Format Item
 - 2.4.3 Structure Declaration and the LIKE Attribute
- 2.5 Instructions
 - 2.5.1 Pointer
 - 2.5.2 Operand
 - 2.5.3 Data Transfer
 - 2.5.4 Stack Utility, Logical and Shift
 - 2.5.5 Decision
 - 2.5.6 Utility
 - 2.5.7 List Processing

Date: 8/4/71
Section: 2 Outline
Page : 1 of 2
Revision:
IBAL MANUAL

- 2.5.8 Imprimitive Instruction
 - 2.5.8.1 Asynchronous Information
- 2.5.9 System Instructions
 - 2.5.9.1 Supervisory
 - 2.5.9.2 Interrupt
 - 2.5.9.3 Input/Output
 - 2.5.9.4 Coordinating
 - 2.5.9.5 Timing
- 2.5.10 Arithmetic
- 2.5.11 Pattern Articulation
 - 2.5.11.1 Boole Function
- 2.6 Input/Output
 - 2.6.1 Interrupt Command
 - 2.6.2 Device Dependent Command
 - 2.6.3 Device Independent Command
- 2.7 Directives
 - 2.7.1 Control Directive
 - 2.7.2 Assignment Directive
 - 2.7.3 Source Format Directive
 - 2.7.4 Assembler Directive

Date: 8/4/71
Section: 2 Outline
Page: 2 of 2
Revision:
IBAL MANUAL

1. Introduction

This manual describes IBAL, the Illiac III Basic Assembler Language. It is intended that IBAL will be the lowest level language used for Illiac III programs. With suitable constraints upon the naming of operands, each primitive operation in IBAL directly corresponds to one Illiac III machine instruction. The reader should be familiar with the Illiac III machine instructions and their formats as given in the Illiac III Reference Manual [1].

It is possible to specify information structures more general than those of the machine instructions. A statement in IBAL may be a block, declaration, instruction or directive. Blocks permit the specification of nested structures of statements. Declarations allow specification of generalized tree-structured data sets whose irreducible constituents are the smallest addressable basic elements of the machine. The block structure of IBAL corresponds to the block structure of ALGOL 60 [2] and PL/1 [3]. The declarations of IBAL are based on those given in [4].

In comparison to other programming languages usually classified as "assembler languages", IBAL allows very general data declarations. It is believed that this capability is essential to any programming language adequate to provide a target language for the translation of procedure-oriented languages having the complexity of PL/1.

Date: 3/26/71
Section: Introduction
Page: 1 of 1
Revision:
IBAL MANUAL

2. Definition of the Language

Syntax is described in this report by metalinguistic formulae with a metalinguistic variable on the left-hand side of the symbol "::<=" and a concatenation of metalinguistic variables, symbols or operators on the right-hand side.

A metalinguistic variable has the form: <any character string>. Its values are finite sequences of symbols. The metalinguistic operators used and their meaning follows:

<u>Operator</u>	<u>Meaning</u>
	Logical OR; the element on the left or the right of the vertical stroke is selected.
{ }	Only one of the elements within the brackets is to be selected.
[]	The element within the bracket may occur once or not at all.
_ _	The element within the brackets may occur any finite number of times.
	The element within the brackets may occur any finite non-zero number of times.

Any character in a formula which is not a metalinguistic variable or operator denotes itself.

A syntactically correct program is generated from the basic variable <program> by replacing it with the right-hand side of any formula in which this variable appears as a left-hand member. Then by continued replacement of variables, a string of symbols, all of which denote themselves is obtained.

Date: 2/9/68
Section: 2
Page: 1 of 2
Revision:
IBAL MANUAL

Examples:

$\langle \text{var } 1 \rangle ::= \langle \text{var } 2 \rangle \mid \langle \text{var } 3 \rangle$

generates $\langle \text{var } 2 \rangle$ or $\langle \text{var } 3 \rangle$

$\langle \text{var } 1 \rangle ::= \langle \text{var } 2 \rangle [\langle \text{var } 3 \rangle]$

generates $\langle \text{var } 2 \rangle$, or $\langle \text{var } 2 \rangle$

concatenated with $\langle \text{var } 3 \rangle$.

$\langle \text{var } 1 \rangle ::= \langle \text{var } 2 \rangle \perp \langle \text{var } 3 \rangle \perp$

generates $\langle \text{var } 2 \rangle$ concatenated with an

arbitrary number of $\langle \text{var } 3 \rangle$.

$\langle \text{var } 1 \rangle ::= \langle \text{var } 2 \rangle \parallel \langle \text{var } 3 \rangle \parallel$

generates $\langle \text{var } 2 \rangle \langle \text{var } 3 \rangle$ concatenated with

an arbitrary number of $\langle \text{var } 3 \rangle$.

Date: 2/26/68

Section: 2

Page: 2 of 2

Revision:

IBAL MANUAL

2.1 Terminal Symbols

The terminal symbols in IBAL are all those elements which denote themselves in a syntax formula (and occur only on the right of such formulae). These symbols are thus terminal elements in the language. A terminal symbol may be a basic character, a compound punctuation, or a reserved word.

Date: 2/9/68

Section: 2.1

Page: 1 of 4

Revision:

IBAL MANUAL

2.1.1 Basic Characters

<alphabetic> ::= \$|#|@|A|B|C|D|E|F|G|H|I|J|K|L
 |M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

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

<special character> ::= |=|+|-|*|/|()|,|.|'|%
 |;|:|_|&|#|>|<|_|?

<blank> ::= single space

<alphanumeric> ::= <alphabetic> | <digit>

<basic character> ::= <alphanumeric> | <special character> | <blank>

The character set is the same as the 60 character set of PL/1 and is given in Appendix 1. There are 29 characters defined as alphabetic characters: Letters A through Z and the three characters \$, @, and #. Decimal digits constitute the ten numeric characters. Finally, the character set is supplemented by 21 special characters.

The language character '|' is denoted by '#' in the syntactic formulae to distinguish it from the metalanguage character '|'.

2.1.2 Compound Punctuations

A compound punctuation is a predefined pair of special characters appearing immediately adjacent to each other. All the compound punctuations are listed below:

comment delimiters:	/*
	*/
PR allocation operator:	->
relational operators:::	<=
	>=
	=
	<
	>

2.1.3 Reserved Words

Reserved words are predefined strings of immediately adjacent alpha-numeric characters. A reserved word may only be used in those places in the program where it is explicitly permitted by a syntactic formula. All the reserved words in IBAL are listed in Appendix 2.

Examples:

```
BEGIN PROCEDURE PUSH END
```

Date: 8/4/70
Section: 2.1
Page: 4 of 4
Revision:
IBAL MANUAL

2.2 Basic Elements

2.2.1 Comments and Blanks

`<comment> ::= /* <any basic character string not containing */> */`

A comment is a string of basic characters not including `*/` which is delimited by `/*` and `*/`. Comments are reproduced on the listing but otherwise have no effect on the program.

For simplicity, the syntactic formulae in this manual do not specify where comments or blanks may occur. The syntactic rules concerning the use of comments and blanks are given below. These rules apply to all places in a program except within a character constant.

IBAL is a free format language. This means that wherever one blank is allowed, any number of blanks may be used. A comment may be used wherever a blank is permitted.

A blank is not permitted within a compound punctuation, an identifier, or within a construct made up of alphanumeric characters.

A blank is required to separate two alphanumeric constructs.

A blank is permitted in all other places.

Date: 8/4/70
Section: 2.2
Page: 1 of 10
Revision:
IBAL MANUAL

2.2.2 Identifiers

`<identifier> ::= <alphabetic> | {<alphanumeric>[-] |`

An identifier is a string of maximum length 31 over the set of alphanumeric characters and the break character. The first character must be alphabetic. The syntactic class, identifier, includes the following language elements:

- label
- name
- reserved word
- symbol
- parameter
- event name
- task name
- entry name

2.2.3 Unsigned Integers

`<unsigned integer> ::=    <digit>`

An unsigned integer is a string of decimal digits of maximum length 5 in the range 0 to 65,535 ($2^{16}-1$).

Examples:

Identifiers: BA13D #5B A_B_C
ABC is one identifier
AB C are two identifiers
X+Y is equivalent to X + Y
/* is a compound punctuation
/ * are two operators
26A is not an identifier
26 is an unsigned integer

Date: 2/9/68
Section: 2.2
Page: 3 of 10
Revision:
IBAL MANUAL

2.2.4 Constants

$\langle \text{number type} \rangle ::= S|L|F|D$
 $\langle \text{hexadigit} \rangle ::= \langle \text{digit} \rangle | A|B|C|D|E|F$
 $\langle \text{bit} \rangle ::= 0|1$
 $\langle \text{sign} \rangle ::= +|-$
 $\langle \text{exponent} \rangle ::= E [\langle \text{sign} \rangle] \langle \text{unsigned integer} \rangle$
 $\langle \text{scale field} \rangle ::= S [\langle \text{sign} \rangle] \langle \text{unsigned integer} \rangle$
 $\langle \text{decimal number} \rangle ::= [\langle \text{sign} \rangle] \{ \langle \text{unsigned integer} \rangle [.]$
 $\quad \quad \quad \langle \text{unsigned integer} \rangle \} | \langle \text{unsigned integer} \rangle$
 $\quad \quad \quad [\langle \text{exponent} \rangle]$
 $\langle \text{binary number} \rangle ::= _ _ \langle \text{bit} \rangle _ _ [\langle \text{scale field} \rangle]$
 $\langle \text{hex number} \rangle ::= _ _ \langle \text{hexadigit} \rangle _ _ [\langle \text{scale field} \rangle]$
 $\langle \text{character string} \rangle ::= \langle \text{string of basic characters with ' occurring}$
 $\quad \quad \quad \text{only in adjacent pairs} \rangle$
 $\langle \text{constant data bits} \rangle ::= \langle \text{character string} \rangle '$
 $\quad \quad \quad | \langle \text{binary number} \rangle _ , \langle \text{binary number} \rangle _ 'B$
 $\quad \quad \quad | \langle \text{hex number} \rangle _ , \langle \text{hex number} \rangle _ 'X$
 $\quad \quad \quad | \langle \text{decimal number} \rangle \langle \text{number type} \rangle$
 $\langle \text{constant flag bits} \rangle ::= ' _ _ \langle \text{bit} \rangle _ _ 'F$
 $\langle \text{constant} \rangle ::= \langle \text{constant data bits} \rangle [\langle \text{constant flag bits} \rangle]$
 $\quad \quad \quad | \langle \text{constant flag bits} \rangle$

A constant specifies a character string or a hexadecimal, binary, or decimal number to be converted to the Illiac III internal representation. The number of bytes used for any constant is determined and the binary representation of the constant is placed in the data bits of the bytes. The constant flag bits which are specified are placed in the flag bits of the bytes. If the specification of either the constant data bits or the constant flag bits are omitted, then these bits are set to zero.

The constant flag bits which are specified are used repeatedly from left to right to set the flags in all the bytes used by the immediately preceding constant data bits.

Date: 2/9/68
Section: 2.2
Page: 4 of 10
Revision:
IBAL MANUAL

Examples:

constant	bytes used	flag bits
'ABC '101' F	3	1 0 1
'98A0'X '01' F	2	0 1
'AJKGLSP' '10' F	7	1 0 1 0 1 0 1
'0A1F24'X	3	0 0 0

A character string constant consists of a string of basic characters, with quotation marks occurring only in adjacent pairs, delimited by a quotation mark on the left and on the right. Each pair of quotation marks is interpreted as one quotation mark to be placed in the store. A single quotation mark is interpreted as the delimiter terminating the character string. The eight bit code for each character, including blanks, is placed in each byte in the store. A comment that is inserted within a character string will not be recognized as a comment but will be considered to be part of the character string data.

A binary constant or a hexadecimal constant consists of a string of binary numbers or hexadecimal numbers separated by commas, enclosed in quotation marks, and followed by B or X respectively. A binary number is a string of bits followed by an optional scale field. A hexadecimal number is a string of hexadigits followed by an optional scale field. The scale field specifies a decimal integer. A positive integer adds binary or hexadecimal zeros to the right of the number, and a negative integer adds binary or hexadecimal zeros to the left of the number. The number is transformed to an unscaled number before the binary representation is placed in the store. If an unscaled binary number does not consist of a multiple of 8 bits, then zeros are inserted to the left of the number until it does. Two hexadigits in a hexadecimal number correspond to one byte in the store. If an unscaled hexadecimal number consists of an odd number of hexadigits, an additional zero hexadigit is added to the left end. The 4-bit code for each hexadigit is placed in the store.

A decimal constant consists of a decimal number followed by a letter indicating the number type. A decimal number may contain a decimal point, and may be preceded by a sign. Decimal numbers may be followed by an exponent which represents a decimal integral power of 10 by which the number is multiplied before conversion.

Date: 8/4/70
Section: 2.2
Page: 5 of 10
Revision:
IBAL MANUAL

Number type D specifies that the decimal number is to be converted to the double-word decimal representation. The decimal point and exponent are ignored. The leftmost byte contains the 4-bit sign code. Then up to 14 most significant digits are right justified within the double word and padded with zeros on the left, if necessary, to fill seven bytes.

Number types S or L specify the short (half-word) or long (full-word) fixed point binary representation. The decimal number is converted to a decimal integer by rounding and truncating if any fractional part remains after adjustment by the exponent. Then the decimal integer is converted to the 2-byte or 4-byte binary integer representation.

Number type F specifies the double-word floating point binary representation. The decimal number is converted to a normalized binary fraction. If necessary, rounding and truncation are performed on the rightmost portion of the fraction. Then the normalized double-word floating point form is placed in the store.

Examples:

Character string constants:	'ABD' 'E?+' '* '''
binary constants:	'101'B '11001S3'B '00,01,10,11'B
hexadecimal constants:	'1A3,4FB'X 'ACB S-1'X '12'X
decimal constants:	253.94 D .25E2 S
	-4.3E-6F 68 S
	1000 L

2.2.5 Literals

`<literal> ::= =<constant>`

A literal specifies a program constant in a decimal, binary, hexadecimal, or character form. When a literal is used as operand, the constant is assembled in an area of fast storage called the literal pool and a pointer to the literal replaces the literal in the program. If identical literals are defined within the same program, the pointers are the same. The literal pool is considered as data which is global to the program. The method of specifying a literal is by an equal sign preceding a constant.

Date: 2/9/68
Section: 2.2
Page: 7 of 10
Revision:
IBAL MANUAL

2.2.6 Self-Defining Values

```
<self-defining value> ::= <unsigned integer>
                        | '<binary number>' B
                        | '<hex number>' X
                        | '<character string>' C
```

A self-defining value is used alone or in an expression to denote a positive integer value. An unsigned integer expresses a decimal value. A binary number expresses a binary value. A hexadecimal number expresses a hexadecimal value. A character string expresses the binary value obtained by converting each character to its 8-bit code. The value is evaluated modulo 2^{16} . Thus the integer value is in the range 0 to $2^{16}-1$.

Examples:

self-defining value	binary equivalent
9	1001
'11011'B	11011
'AlB'X	101000011011

Date: 2/9/68
Section: 2.2
Page: 8 of 10
Revision:
IBAL MANUAL

2.2.7 Expressions

```
<label> ::= <identifier>
<symbol> ::= <identifier>
              | <register symbol>
<register symbol> ::= ØS|PRO|PR1|PR2|PR3|PR4|PR5|PR6|PR7|PR8|PR9
                    |PR10|PR11|PR12|PR13|PR14
<primary> ::= <self-defining value>
              | <symbol>
              | <label>
              | (<constant expression>)
<term> ::= [ <term> {*/} ] <primary>
<constant expression> ::= [ [ <constant expression> ] { + | - } ] <term>
<integer expression> ::= <constant expression>
                        | <operand>
```

A symbol is an identifier which is assigned a value by the EQU directive or is a predefined register symbol. A label is an identifier whose value is determined by its position within the program segment.

Expressions in IBAL are used only to obtain an integral value. The value of a constant expression can be determined at compile time. The absolute value of a constant expression will be evaluated modulo 2^{16} . Thus a value in the range $-(2^{16}-1)$ to $(2^{16}-1)$ can be represented. In a constant expression the predefined pointer register symbols PRO, PR1, ..., PR14 have the integral values 0, 1, ..., 14 respectively.

The operations performed in evaluating an expression are unary negation and binary addition, subtraction, multiplication, and division. These operations are performed on the long fixed point binary representation of the values of the primaries in the expression. The precedence of operators is: first unary + and - , second binary * and / , third binary + and - . Expressions within parenthesis have the highest precedence. In the case of equal precedence operators on the same parenthesis level, the leftmost operation is performed first.

Date: 2/9/68
Section: 2.2
Page: 9 of 10
Revision:
IBAL MANUAL

An integer expression is a constant expression or is itself an operand (operands are defined in Section 2.5.2). If an integer expression is an operand, then the value specified is the 16-bit positive integer at the operand address. If the operand as an integer expression is itself a constant expression, then the constant expression must contain one of the predefined pointer register symbols; otherwise, the value of the integer expression will be interpreted as the value of the constant expression. For example, 2 specifies the integer value 2, while PR2 specifies the integer value at the location pointed to by the pointer register named 2.

Examples:

symbols:	A, S1, PR3 , PR12
constant expressions:	A+'2B'X, (A-S1)/3 , PR3 + PR12
integer expressions:	A + 2, PR3, F.M(2,3)

2.3 Program Structure

< parameter> ::= < identifier>

< statement> ::= < declaration>

| < instruction>

| < directive>

| < block>

| < boole function>

| < label> : < statement>

<block> ::= BEGIN ; | <statement>; | END

| PROCEDURE [SEGMENT|<storage attribute> |]

[(<parameter>|, <parameter>|)];

|<statement>; | END

|<label> : <block> <label>

<program> ::= <block>;

Date: 7/30/71

Section 2.3

Page: 1 of 5

Revision:

IBAL MANUAL

2.3.1 Blocks

An IBAL program is a block. A block is a series of statements separated by semicolons and delimited by BEGIN and END or by PROCEDURE and END. A statement is a declaration, an instruction, a directive, or is itself a block. Thus, in general, a program is a nested structure of blocks. The outermost block will be said to be at level one. Any block which is a statement in a block at level n is at level $n + 1$.

Any statement may be named by preceding it by a label and a colon. For clarity, the same label placed before a block may be placed directly after the block END. The label following a block END will cause the labeled block as well as all blocks within the labeled block to be terminated. For example, if block A surrounds block B, then "END A" is equivalent to "END B; END A". The label of the block at level one is called the program name. Labels on a block delimited by BEGIN and END are block names, or if delimited by PROCEDURE and END are procedure names. Labels on an ENTRY directive are entry names and labels on the EQU directive are symbols. All other statement labels are undifferentiated.

Blocks delimited on the left by BEGIN are executed in their order of appearance.

Date: 2/9/68
Section: 2.3
Page: 2 of 5
Revision:
IBAL MANUAL

2.3.2 Procedures

Blocks delimited on the left by PROCEDURE are called procedures and are executed as subroutines upon execution of the CALL operator.

A procedure is a subroutine which receives special treatment by the system since it has been explicitly declared by the delimiters PROCEDURE and END. It is a block, and is executed only where the CALL operator appears. Operands following the CALL operator may be used to specify procedure operands.

A list of parameters may appear after the word PROCEDURE. Parameters within the procedure block will be assigned to pointer registers as determined by their order of appearance within the parameter list.

A procedure must be named by a label. The first label is the procedure name and specifies the primary entry point. Other labels specify additional entry names for the procedure.

Date: 8/4/70
Section: 2.3
Page: 3 of 5
Revision:
IBAL MANUAL

2.3.3 Scope

All labels and names used in a block must be declared either in the block or in a surrounding block. A label is declared implicitly by its position in front of an item. A name which is not a label is declared explicitly in a declaration. The scope of a label or name is that region of program over which it is defined. A label or name is local to the block in which it is declared, and its scope is this entire block, excluding any contained blocks where the same name or label is declared. The label or name is global to all other contained blocks. For further explanation and examples see Section 8 of [5].

Date: 2/9/68

Section: 2.3

Page: 4 of 5

Revision:

IBAL MANUAL

2.3.4 Storage Allocation

Block structure facilitates dynamic storage allocation. At any one instance in the execution of the program, storage is allocated only for data items which are local or global to the block being executed. Thus, in their allocation, data items correspond to variables of ALGOL 60 and to the automatic storage class of PL/1.

Date: 2/9/68
Section: 2.3
Page: 5 of 5
Revision:
IBAL MANUAL

2.4 Declarations

< declaration> ::= { DECLARE [SEGMENT | < storage attribute> |] | STRUCTURE }
 < format item> | { , | * } < format item> |

< simple format item> ::= [< level>] [[*] < name>] [< attributes>]

< level> ::= < unsigned integer>

< name> ::= < identifier>

< format item> ::= < simple format item> | < conditional format item>

A declaration, prefixed by DECLARE, is the general means to define a collection of data sets. Each data set may have a general tree structure. The tree is declared in a top down manner. That is, the first identifier declared is a unique name and is said to be at level one. Branches leading from this top node lead to nodes at level two. Additional branching increases the level until a terminal node is reached.

Each format item in the declaration specifies a node in the structure. The level number is either specified explicitly by a positive integer or may be omitted if the node is at level one. A series of format items separated by commas specify a complete access structure for a data set. The format items must be given in the order that would be obtained if the tree structure were followed in a top down manner, always taking the leftmost branch until a terminal node is reached. Alternate access structures for the same data may be specified by separating two adjacent specifications by an OR ' | '. The unique level one name will determine which access structure is used in any one reference.

Following the level number, each format item contains an optional name followed by the node attributes. The name specifies the node name. It must be present for all nodes at level one and it must be present if the dimension attribute is used. If an asterisk precedes a node name, the named item is optional and need be provided only if a reference is made to the name.

Date: 7/26/71
Section: 2.4
Page: 1 of 11
Revision:
IBAL MANUAL

2.4.1 Attributes

```
<attributes> ::= <nonterminal node attributes>
                | <terminal node attribute>
<nonterminal node attributes> ::= [( <bounds> | , <bounds> | ) ]
                                   [ { STRING | VECTOR } ]
<terminal node attribute> ::= <cell size> | INITIAL( <constant list> )
                               | LIKE <name>
<cell size> ::= B | H | W | D
<bounds> ::= [ <lower bound> : ] <upper bound>
              | > <bound>
              | < <bound>
              | *
<bound> ::= <integer expression>
<lower bound> ::= <integer expression>
<upper bound> ::= <integer expression>
<constant list> ::= [( <unsigned integer> ) ] <cspec>
                    | , [( <unsigned integer> ) ] <cspec> |
<cspec> ::= <constant> | *
```

The nonterminal node attributes are the dimension attribute, the vector attribute and the string attribute.

The dimension attribute specifies that this node is an array of elements of the type specified by the substructure of this node. The dimension attribute must immediately follow the node name. The size of each dimension of the array is specified by bounds separated by commas and enclosed in parentheses. The bounds may specify a lower bound and an upper bound on the index to be used in referencing the array. If the lower bound is omitted, it is assumed to be one. The bounds may also specify that the exact size of this dimension of the array cannot be determined until execution. In some cases, the array is dynamic within the block in which it is declared. The following three cases specify dynamic bounds:

> <bound>	bound specifies minimum size
< <bound>	bound specifies maximum size
*	no estimates on the size

Date: 8/5/70
Section: 2.4
Page: 2 of 11
Revision: IBAL MANUAL

Each bound is an integer expression. If a variable integer expression is used, the expression is calculated at the time that storage allocation is done for the block. Accordingly, a variable in an integer expression of a declaration must be global to the block in which it occurs.

The attributes VECTOR or STRING specify the way in which the elements at the next higher level are linked, i.e. whether the present node is a vector or a string of the nodes immediately under it. If neither VECTOR or STRING is specified, VECTOR is assumed. Referencing an ordered list of items will be more efficient if they are declared as vector elements, while additions and deletions of information can be performed more efficiently for lists declared as strings. Note that any node in the file may be a string or a vector. Consequently any item in a string or vector can subsume other vectors or strings.

The terminal node attribute specifies the cell size of the data element at this location in the file, or constants to initially occupy locations in the file, or the LIKE attribute. The cell sizes are 1, 2, 4, or 8 bytes specified by B, H, W, or D respectively. The LIKE attribute is explained in section 2.4.3. The INITIAL attribute specifies constant values to be assigned to data elements when storage is allocated for the file. An element which is not a member of an array must have only one constant in the constant list. Constants in the constant list for an array are assigned in a row-major order to the array elements. An unsigned integer, *i*; preceding a constant is equivalent to repeating the constant in the list *i* times. An asterisk specifies that no initialization is done for this element.

Date: 8/5/70
Section: 2.4
Page: 3 of 11
Revision:
IBAL MANUAL

2.4.2 Conditional Format Item

```

< conditional format item> ::= < if clause> < simple format item>
                                ELSE < simple format item>
< if clause> ::= IF < Boolean expression> THEN
< Boolean expression> ::= < Boolean term> | < Boolean term> |
< Boolean term> ::= < Boolean factor> | & < Boolean factor> |
< Boolean factor> ::= (< Relation>) | < Logical value>
                                [ NOT ] ( < Boolean expression> )
< Logical value> ::= FALSE | TRUE
< relation> ::= < integer expression> < relational operator>
                                < integer expression>
< relational operator> = > | < | >= | <= | <= | <=

```

A conditional format item is a format item with the form given above. At the time when storage is allocated, the Boolean expression is evaluated. If the value of the expression is T, true, then the first format item is chosen for this element in the data set, otherwise the second format item is chosen.

A Boolean expression is made up of the logical operators: NOT "NOT", AND "&", and OR "|"; and the relations: Less than "<", Less than or equal "<=", Equal "=", Greater than ">", Greater than or equal ">=", unequal "≠", not less than ">", and not greater than "<>" between integer expressions. The relations are the primaries in the Boolean expression and yield a value T, true or F, false. The meaning of the unary and binary logical operators is given in the following table:

b1	F F T T
b2	F T F T
---	---
NOT b1	T T F F
b1 & b2	F F F T
b1 b2	F T T T

Date: 7/26/71
 Section: 2.4
 Page: 4 of 11
 Revision:
 IBAL MANUAL

The order of evaluation of the Boolean expression is from left to right with items within parenthesis being evaluated first. The precedence of Boolean operations is:

First:	Integer expressions and $\neg$
Second:	Relational operators
Third:	$\&$
Fourth:	$\#$

These precedences are consistent with the operators of PL/1.

Date: 3/30/71
Section: 2.4
Page 5 of 11
Revision:
IBAL MANUAL

2.4.3 Structure Declaration and the LIKE Attribute

The STRUCTURE declaration is syntactically equivalent to the DECLARE declaration (apart from the replacement of the reserved word DECLARE by the reserved word STRUCTURE), but causes no storage to be allocated. The structure defined by the structure declaration may be symbolically inserted into any declaration within its scope. This is done by using the name of the top node of the STRUCTURE declaration following the LIKE terminal node attribute in a DECLARE declaration.

Examples:

The declaration of Example 1, given below, defines a static data set. Although storage is allocated for the data each time the block which contains the declaration is entered, the amount of storage necessary for the data never changes. All the elements are vector elements.

Example 2 defines a dynamic data set of all vector elements. N and L may vary each time the block containing the declaration is entered.

The data set of Example 3 consists of both vector and string elements. Branches leading to string items are dotted. The contents of the strings, in this example, are all character constants.

Date: 2/9/68
Section: 2.4
Page: 6 of 11
Revision:
IBAL MANUAL

Example 1 - Static Data

Data declaration:

DECLARE 1 F, 2 W, 2, 3 A1, 4 B, 4 H, 3 B, 3 A2, 4 H, 4 B,
4 W, 3 D, 2 B

Data structure:

"---integer" specifies the level

Date: 2/9/68
Section: 2.4
Page: 7 of 11
Revision:
IBAL MANUAL

Example 2 - Dynamic Data

Data declaration:

DECLARE 1 F, 2 A1(N), 3 B, 3 H, 2 W, 2 A2(L), 3 H, 3 B, 3 W

Data structure:

Example 3 - Usage of a Structure

1 S,
 2 T,
 3 U B,
 3 V B,
 2 W H;

DECLARE 1 A,
 2 B H,
 2 C,
 3 DISLIKES,
 3 E H,
 2 F W;

Declare must be in the scope of STRUCTURE DEFINITION

Date: 8/2/71
 Section 2.4
 Page 9 of 11
 Revision:
 IBAL MANUAL

Example 4

Data declaration:

```
DECLARE 1 A, 2 C STRING,  
        3 INITIAL ('B'),  
        3 D, 4 W, 4 W,  
        3 E STRING, 4 INITIAL ('IS'), 4 INITIAL ('=PQ'),  
        3 INITIAL ('A'),  
        2 W
```

Data structure:

Example 5

```
DECLARE 1 A (M:N) STRING, 2 V (P), 3 W ;
```

A is a string of N-M+1 components, each one of which is a vector of P elements, each of which is a word.

Example 6

```
DECLARE 1 S (>M) STRING, 2 INITIAL (' ');
```

S is a string of at least M components, each of which is initially the blank character.

Example 7

```
DECLARE 1 STACK(*), IF RUN=1 THEN 2 W ELSE 2 H ;
```

STACK is a vector with an arbitrary number of components, each one of which is a word or a halfword.

Example 8

```
DECLARE 1 A(M, N), 2 D | 1 AA (M, 2*N), 2 W ;
```

One array is defined with two access structures. A is a M by N double word matrix structure. AA is a M by 2*N word matrix structure.

Date: 8/5/70
Section: 2.4
Page: 11 of 11
Revision: 1
IBAL MANUAL

2.5 Instructions

```
<instruction> ::= <imprimitive instruction>
                  | <basic data transfer>
                  | <logical>
                  | <string and list processing>
                  | <segment administration>
                  | <arithmetic>
                  | <decision>
                  | <system>
                  | <input/output>
                  | <pattern articulation>
                  | NOP
```

The instructions in IBAL correspond to the Illiac III machine instructions. The form of an instruction is a mnemonic specifying the operation, followed by operand specifications separated by commas. In most cases all the information preceding the first comma is assembled into one mnemonic byte, and each operand specification is assembled into one byte (short) or three bytes (long).

Date: 3/30/71
Section: 2.5
Page: 1 of 17
Revision:
IBAL MANUAL

2.5.1 Pointer

<subscript> ::= <integer expression>

<pointer> ::= <name> [(<subscript>|, <subscript>|)]
|. <name>[(<subscript>|, <subscript>|)]|

A pointer is the means of referencing data sets declared in a DECLARE declaration. A pointer may occur as an operator or as an operand in an instruction.

A pointer specifies one node in a tree structured data set. The first name must be the unique file name. If subscripts follow the name, the name is called a subscripted name. The first subscript specifies the index of the branch leaving the named node going to a node at the next lower level. Additional subscripts specify branches leading to nodes at lower levels. If other names (separated by periods) follow the first name, the pointer is called a qualified name. The name following a period must be unique within the substructure of the file pointed to by the preceding name. This name with its subscripts (if any) may lead to a lower level node. Subscripts or names may be continued in this manner until a terminal node is reached. Since node names are optional except at level one, subscripts may be used to select branches at any level within the structure, not just within arrays. Subscripts however must be present to select branches within an array. Subscripts are used from left to right. When an array of K dimensions is reached, the next K subscripts are used for that array, regardless of whether the K subscripts are properly within one subscript list, exactly one subscript list, or one or more subscript lists (i.e. interleaved between names). Thus a pointer may be denoted by a concatenation of names with subscripts. Any one name may not be unique, but the complete subscripted qualified name must uniquely specify a node in a declared data set.

A pointer is transformed into an operator or operand by identifying a pointer register which points to the leftmost byte of the named node. This pointer register allocation algorithm is discussed in Section 4.

Date: 2/9/68
Section: 2.5
Page: 2 of 17
Revision:
IBAL MANUAL

A pointer used as a procedure operand may specify a nonterminal node in the data set. This indicates that the data subset subsumed by the node must be available for access as a procedure operand. In this case the pointer register associated with the operand may vary over the entire subset.

In example 1 of Section 2.4, $F(2,3,1)$ or $F(2).A2(1)$ are pointers to the ninth byte of F . Since a subscript may be an operand, pointers may be used recursively. Thus, in Example 2 of Section 2.4, $F(3,F.A1(1,2),2)$ points to the second element of the element of $A2$ given by the value of the 16 bit positive integer at the second byte of $A1$.

Examples:

$A(1).B.C$

$A(1,2)$

B

$D . E$

Date: 3/7/68
Section: 2.5
Page: 3 of 17
Revision:
IBAL MANUAL

2.5.2 Operand

```
<modifier> ::= <constant expression>
<modification operator> ::= + | - | =
<tag> ::= <symbol> | <self-defining value>
<pointer specification> ::= <tag> [*]
                                | <tag> <modification operator> <modifier>
                                | <pointer>
<operand> ::= [*] <pointer specification> [*]
                | <literal>
```

An operand may explicitly identify a pointer register name and operations on the named pointer register, or it may identify a pointer, or be a literal. The machine form of an operand will always contain a pointer register name.

A tag is a symbol or a self defining value, evaluated modulo 2^4 , specifying a 4-bit pointer register name.

An asterisk following the tag specifies indirect replacement if no modifier is present.

A modifier is a constant expression specifying a 16-bit pointer value. If the constant expression contains no register symbols, it is a direct modifier and will be evaluated modulo 2^{16} . If the constant expression contains a register symbol, it is an indirect modifier and will be evaluated modulo 2^4 . The value of the predefined register symbol $\emptyset S$, which specifies the operand stack, is '1111' B when it is used in a modifier expression.

The modification operator specifies one of the following three possible operations on the named pointer register:

+	addition
-	conditional subtraction
=	replacement

Date: 7/30/71
Section: 2.5
Page: 4 of 17
Revision:
IBAL MANUAL

Pushing or popping of the named pointer register is indicated by a slash before or after the pointer specification respectively.

When a literal is used as an operand, a pointer register is allocated from those assigned to the base field of the literal pool.

Date: 2/26/68
Section: 2.5
Page: 5 of 17
Revision:
IBAL MANUAL

2.5.3 Data Transfer

<register variant> ::= F|L|V|R

<field designator> ::= <cell size> | <register variant>

<basic data transfer> ::= <type 1> [<field designator>], [...] <operand>
| <type 2> [<field designator>]
, [...] <operand>, [...] <operand>

<type 1> ::= PUSH|POP|LD|ST|SET|RESET|TEST|TESTM

<type 2> ::= ASSIGN

The basic data transfer instructions (single cycle) operate on either a cell size in core or a full word register. An operand specifies a cell in core by naming a pointer register. If a period precedes the operand it is an immediate operand which indicates that the contents of the named pointer register is the operand. The field designator specifies the actual cell size in core or the part of the register to be operated on, as given below:

field designator	meaning
B	1 byte (Byte)
H	2 bytes (Halfword)
W	4 bytes (Word)
D	8 bytes (Doubleword)
F	4 flag bits (Flags)
L	left 16 data bits (Link)
V	right 16 data bits (Value)
R	all 36 bits (Register)

The cell size or register variant may only be omitted if the instruction is within the scope of a SETCS or SETRV directive respectively.

2.5.4 Stack Utility, Logical and Shift

<count> ::= <operand>

<logical> ::= <type 3> [<cell size>]

 | <type 4> [<cell size>], <count>

 | <type 5> [<cell size>], [, <count>], [, <operand>]

<type 3> ::= ONE|ZERO|NOT|COUNT|BIT|AND|OR|XOR|EQV|CPRL|XCH|DUP|SLUFF

<type 4> ::= LS|RS

<type 5> ::= SCAN|SCANM

Logical instructions of type 3 include stack utility instructions;
Exchange, Duplicate and Sluff.

Type 4 are shift instructions.

Type 5 are field comparison instructions.

Date: 7/26/71

Section: 2.5

Page: 7 of 17

Revision:

IBAL MANUAL

2.5.5 Decision

`<decision> ::= <type 12> <1,condition> 1, 'label'`

`<type 12> ::= IF|IFN`

`<condition> ::= <|=|>|FM|OV|CS`

Each condition specified in a decision instruction sets the corresponding bit in the mask byte of the instruction.

CS = Conditional Subtraction

OV = Overflow

FM = Flag Match

Date: 7/26/71
Section: 2.5
Page: 8 of 17
Revision:
IBAL MANUAL

2.5.6 Utility

$\langle \text{segment administration} \rangle ::= \langle \text{type 9} \rangle, \langle \text{operand} \rangle, \langle \text{operand} \rangle$
 $\quad \quad \quad | \langle \text{type 10} \rangle$

$\langle \text{type 9} \rangle ::= \text{LBR} | \text{SBR}$

$\langle \text{type 10} \rangle ::= \text{LAR} | \text{SAR} | \text{LPC}$

A base register points to the beginning address of a segment and works as a data base for that segment.

2.5.7. List Processing

```

<string and list processing> ::= <type 6> [, <count> ], <operand>
                                | <type 7> , <operand>
                                | <type 8> [, <count> ], <operand> , <operand>
<type 6> ::= PUSHF | PØPF | PUSHFR | PØPFR | PACK | UNPACK
<type 7> ::= SL | SR | GETC | PUTC | INCL | INCR | DECL | DECR
<type 8> ::= EDIT | TRANS | MØVE

```

Type 6 is part of an instruction set used in MULTIPLE cycle data transfers. The first 4 are operand stack field transfer instructions and the last 2 are binary conversion instructions.

Type 7 are list processing instructions and Type 8 are string manipulation instructions.

Date: 3/7/68
Section: 2.5
Page: 10 of 17
Revision:
IBAL MANUAL

2.5.8 Imprimitive Instruction

```
<entry name> ::= <label>
<operator mnemonic> ::= GØ TØ | CALL | EXECUTE
<operator> ::= <operator mnemonic> <pointer specification>
               | GØ TØ <label>
               | CALL <entry name> <asynchronous information>
               | EXECUTE <label>
               | SPECIFY
<imprimitive instruction> ::= <operator> ┘, <operand> ┘ | EXIT
```

An imprimitive instruction causes operations on pointer registers and usually a transfer of control. The operator mnemonics correspond to the similarly named machine instructions. The operator pointer specification may be a label in the GØ TØ and EXECUTE operator, and may be an entry name in the CALL operator.

Date: 8/5/71
Section: 2.5
Page: 11 of 17
Revision:
IBAL MANUAL

2.5.8.1 Asynchronous Information

```
<asynchronous information> ::= [ TASK [ (<task name>)] ]  
                                [ EVENT (<event name>) ]  
                                [ PRIORITY (<integer expression>) ]  
  
<task name> ::= <identifier>  
<event name> ::= <identifier>
```

The occurrence of any asynchronous information with the CALL operator will inform the execution time system that the calling program and the called procedure may be executed concurrently. Programs or parts of programs which may be executed simultaneously are called tasks. These three types of information are called the task, event, and priority options.

The event option defines an event name which may be used to test whether or not the called task is completed. The event name is a pointer to a binary value which is set to '0'B on execution of the call and set to '1'B on completion of the called task. With the above definition, we can have different event completion in different stages of Task being processed.

The priority option sets the priority of the procedure, relative to the calling task, as the value of the integer expression relative to one. A negative value inverts the ratio.

The task option need not be given if the event or priority options are given unless a task name is desired.

2.5.9 System Instructions

$\langle \text{system} \rangle ::= \langle \text{supervisory} \rangle | \langle \text{interrupt} \rangle | \langle \text{input-output} \rangle |$
 $\langle \text{coordinating} \rangle | \langle \text{timing} \rangle$

2.5.9.1 Supervisory

$\langle \text{supervisory} \rangle ::= \{ \text{SVC} | \text{SVR} | \text{ACTP} | \text{LTR} | \text{STR} \}$
 $| \text{SLEEP}, \langle \text{operand} \rangle$
 $| \{ \text{RNAM} | \text{RESU} \}, \langle \text{operand} \rangle, \langle \text{operand} \rangle$

2.5.9.2 Interrupt

$\langle \text{interrupt} \rangle ::= \text{INRT} | \text{SIM}, \langle \text{operand} \rangle$

2.5.9.3 Input /Output

$\langle \text{input/output} \rangle ::= \text{SIO} | \text{HIO} | \text{LIBR}, \langle \text{operand} \rangle$

2.5.9.4 Coordinating

$\langle \text{coordinating} \rangle ::= \text{WHO} | \text{INCK}, \langle \text{operand} \rangle$
 $| \text{LINK}, \langle \text{operand} \rangle, \langle \text{operand} \rangle$

2.5.9.5 Timing

$\langle \text{timing} \rangle ::= \text{RDCLK} | \text{STTIM} | \text{RDTIM}$

2.5.10 Arithmetic

<number type> ::= S|L|F|D

<arithmetic> ::= <type ll> [<number type>]

<type ll> ::= NEG|ABS|MNS|TA|ADD|SUB|MULT|DIV|CPRA|POLY|CVD|CVF|CVL

The arithmetic instructions specify arithmetic operations on one of four possible number types:

number type	meaning
S	Short Fixed point (halfword)
L	Long Fixed point (word)
F	Floating point (doubleword)
D	Decimal (doubleword)

The number type may be omitted only if the instruction is within the scope of a SETNT directive.

The TP is capable of doing some of these arithmetic operations with certain data types.

2.5.11 Pattern Articulation

<pattern articulation> ::= <0-plane>

| <1-plane>

| <2-plane>

| <3-plane>

| <multi-plane>

| < border>

<0-plane> ::= ~~TOPOLOGY~~ [{RECTANGULAR|HEXAGONAL}]

[{PLANAR|~~TOROIDAL~~}] | RESUME | RESTART | SETORG, < operand>

<1-plane> ::= {CLEARP|SETP}, <ID variant>, <plane name>

| {TESTP|TESTB|LISTI|ERASEP}, <plane name>

| SHIFT, <plane name>, <displacement>

| {TALLY|TALLYH}, <plane name>, <direction list>

| {AREA|LIST|LISTSZ|LISTLZ}, <plane name>[, <count>]

| {PLOT|PLOTSZ|PLOTLZ|PLOTI}, <plane name>, <string name>

| READLZ|RDERLZ, <plane name>

| WRITLZ|WRERLZ, <plane name>

<ID variant> ::= P|E|W|N|S|NE|NW|SE|SW

<plane name> ::= <operand>

<displacement> ::= <operand>

<direction list> ::= <operand>

<string name> ::= <operand>

<2-plane> ::= {COPY|COPYC|PLAND|PLOR|PLNAND|PLNOR|PLEXOR|PLEOV}, <plane name> ,

<3-plane> ::= CONNECT, <plane name>, <plane name> <plane name>

[, <plane name>], <direction list>

<multi-plane> ::= REPLICATE | GATEIA, <IAWORD word>

| B~~OO~~LE, <availability list>, <operand>

<availability list> ::= <constant> | <symbol>

Date: 8/5/71
Section: 2.5
Page: 15 of 17
Revision:
IBAL MANUAL

```

<border> ::= {LOADB|STOREB|PUSHB|POPB} , <gbw>,<string name>
           | MOVEB,<gbw>,<plane name>,<plane name>
<gbw> ::= <constant>|<symbol>

```

The pattern articulation instructions specify operations performed by the pattern articulation unit of the Illiac III.

The default options in the TOPLOGY instruction are RECTANGULAR and PLANAR.

Plane names, displacements, direction lists, and string names are specified by general operands. Availability lists, gatewords, and gbwords are constants. These constants may be specified directly in the instruction or by a symbol which is assigned a value by an assignment directive.

The availability list tells what planes in the Iterative Array may be used as scratch planes in the evaluation.

Date: 7/26/71 Section: 2.5 Page: 16 of 17 Revision: IBAL MANUAL

2.5.11.1 Boole Function

<Boole function> ::= BOOFUN [#]<identifier> = <character string>

An operand phrase in BOOLE instruction is the address of a variable length field which describes in Polish notation, the Boolean function to be executed.

The PAU expects the following form for general boolean function:

```
<function> ::= <term string>;  
<term string> ::= <term> | <term>; <term string>  
<term> ::= <elementary function> | <elementary function> <operator  
string>  
<operator string> ::= <secondary operator> | <secondary operator>  
    <operator string>
```

Here an elementary function is defined as any function meeting both of the following criteria:

- 1) The domain of the function is restricted either to (a) any one plane, or (b) the vertical column passing through bit position 0.
- 2) The function is a simple Boolean sum (OR) or a simple Boolean product (AND) of the individual (true and/or complemented) variables.

If # is not specified before <identifier> then the character string is in the form of familiar Boolean expression as defined in this manual (2.4.2).

For details about implementation see *Illiac III Reference Manual* VOL. II.

Example 1. Boolean Expression

```
BOOFUN P6 = ¬(¬(10)&(12)&(16)|(20))|(34)&(38);
```

Example 2. Canonical form

```
BOOFUN #P6 = [¬(10)&(12)&(16)];[20]|¬;[(34)&(38)];
```

Date: 7/30/71
Section: 2.5
Page 17 of 17
Revision:
IBAL MANUAL

2.6 Input/Output

$\langle \text{command} \rangle ::= \langle \text{interrupt command} \rangle \mid \langle \text{device dependent command} \rangle$
 $\mid \langle \text{device independent command} \rangle$

Input/output commands are used to control the transmission of data between the main store and peripheral devices. Commands are interpreted by input/output processor (IOP) of the Illiac III.

I/O commands are subdivided into three main classes.

2.6.1 Interrupt Command

$\langle \text{interrupt command} \rangle ::= \text{SIO}, \langle \text{channel name} \rangle, \langle \text{pointer value} \rangle$
 $\mid \langle \text{channel name} \rangle$
 $\mid \text{HIO}, \langle \text{program tag} \rangle$
 $\mid \text{LIBR}, \langle \text{BR tag} \rangle, \langle \text{base register contents} \rangle$

$\langle \text{channel name} \rangle ::= \langle \text{constant expression} \rangle$

$\langle \text{pointer value} \rangle ::= \langle \text{constant expression} \rangle$

$\langle \text{program tag} \rangle ::= \langle \text{symbol} \rangle \mid \langle \text{self defining value} \rangle$

$\langle \text{BR tag} \rangle ::= \langle \text{symbol} \rangle \mid \langle \text{self defining value} \rangle$

$\langle \text{base register contents} \rangle ::= \langle \text{constant expression} \rangle$

Channel name specifies one of 8 channels in an IOP, and it is evaluated modulo 8.

A pointer value is a constant expression specifying a command pointer and it is evaluated modulo 2^{16} .

Program tag specifies a program register name and is evaluated modulo 8.

BR tag specifies one of two base registers in an IOP, namely Command Base Register (CBR) or Descriptor Base Register (DBR). By convention the first channel name in SIO is also the program name.

SIO command can be followed by at most two other channel names.

Date: 8/4/71
Section: 2.6
Page: 1 of 3
Revision:
IBAL MANUAL

2.6.2 Device Dependent Command

```

< device dependent command> ::= < DDC mnemonic> | < DDC tag> |,
                                < extended device address>
< DDC mnemonic> ::= READ | READR | SENSE | WRITE | CONTROL | TESTDS | POLL
< DDC tag> ::= RD | DEI | P | PO | R
< extended device address> ::= < rate code>, < channel name>,
                                < device address>
< rate code> ::= < constant expression>
< device address> ::= < constant expression>

```

The mnemonic part of the device dependent command is followed by any permissible combination of DDC tags: RD, DEI, P, PO, R which specify RELEASE DEVICE, DEVICE END INTERRUPT, PROCEED, PARITY OVERRIDE and RESTART respectively. For details see Illiac III Reference Manual III [1].

The rate code is related to actual device rate and is evaluated modulo 8.

As the channel name specifies, one of 16 channels in Illiac III is evaluated modulo 16 in a DDC. Device address is evaluated modulo 28. In this way symbolic representation of channels and devices can be used.

Date: 8/4/71
Section: 2.6
Page: 2 of 3
Revision:
IBAL MANUAL

2.6.3 Device Independent Command

```
< device independent command> ::= < DIC mnemonic>< field designator>,  
                                     < DIC tag>< modification operator>  
                                     < modifier>
```

```
< DIC mnemonic> ::= LOAD|STORE|SETM|RESETM|TESTANY|TESTALL|MODIFY|STOP
```

< DIC tag > ::= < program or channel > , < name > , < register address >

$$\langle \text{program or channel} \rangle ::= p | c$$

```
< name > ::= < identifier> | < self defining value>
```

```
< register address > ::= < identifier> | < self defining value>
```

These commands are usually used for bookkeeping operations, e.g. address modification, loading and storing the contents of a program or channel register.

The mnemonic part consists of the command mnemonic followed by the field designator for the register to be operated on. This field designator is interpreted as for the taxicrinic processor.

A register specification containing a tag part and an operational modifier part follows the mnemonic byte.

P and C specify Program and Channel module respectively. The name of the module is given by the name which is evaluated modulo 8.

Register address specifies the register to be modified and is evaluated module 16. Programmers are not allowed to access registers SDR, CBR, and CSBR.

The modification part is interpreted the same way as in TP and 'must' always be specified.

Date: 8/9/71
Section: 2.6
Page 3 of 3
Revision: IBAL MANUAL

2.7 Directives

```
<directive> ::= <control directive>
                | <assignment directive>
                | <source format directive>
                | <assembler directive>
```

A directive is a statement which may result only in action by the assembler at translation time or may cause code to be executed at run time or both.

2.7.1 Control Directive

```
<control directive> ::= ENTRY
                        | RETURN
                        | WAIT <event name>
```

An ENTRY or RETURN directive may appear only within a procedure. An ENTRY directive specifies an entry point to the procedure in which it occurs. That is, the scope of the label of the entry statement is the same as the scope of the procedure name. An entry name is either an entry label or a procedure name. An unlabeled ENTRY is a dummy item. When a procedure CALL is executed, instruction execution begins at the entry point specified by the entry name and terminates when the procedure END or a RETURN directive is encountered. These are the only valid returns since only they will free the local variables of the procedure block. In all other respects the return is equivalent to an EXIT instruction.

For coordination of tasks, the WAIT directive is provided. Execution of the statement containing the WAIT directive causes the task in which it is located to be halted until the value of the event name equals '1'B.

Date: 8/4/71
Section: 2.7
Page: 1 of 5
Revision:
IBAL MANUAL

2.7.2 Assignment Directive

`<assignment directive> ::= ALLØCATE ll, <pointer> -> <tag> ll
| EQU , <constant expression>`

The ALLØCATE directive is provided for the user to optimize allocation of pointer registers to symbolic pointers. For each pointer named, the pointer value and base are determined and loaded into the pointer register specified by the tag, before the execution of any subsequent statements. This tag may then be used subsequently as a pointer specification to point to the desired data element. In this way the complicated allocation and transformation operations from pointer to tag can be avoided for subsequent references to the same element.

The EQU directive defines a compile time symbol. A label on the EQU statement is defined as a symbol whose value is equated to the value of the constant expression. Symbols may be used as primaries in constant expressions.

Date: 8/4/71
Section: 2.7
Page: 2 of 5
Revision:
IBAL MANUAL

2.7.3 Source Format Directive

<column> ::= <unsigned integer>

<source format directive> ::= LINE [<column>, <column>, <column>]
| LINE FREE

The source format directive defines a line format for the source language. The defined format will be used for all lines following the present LINE directive, up to and including the next LINE directive. Free format is assumed up to the first LINE directive.

The line format is specified by three integers: i, j, k where i, j, k ≤ 80. Each line is interpreted as one statement. Only columns i through k inclusive on a line are assumed to be characters in the program, although the whole line is listed. If column k is blank, it is interpreted as the semicolon ending the statement; otherwise, the next line continues the same statement.

A statement label may appear in column i through j-2. No colon is necessary following the statement label in this position.

Columns j through k-1 are interpreted exactly as in format-free IBAL and may be considered as the operation field of the statement.

If an asterisk appears in column i of a line, then the line is listed but is otherwise ignored by the assembler.

Format:

The LINE directive may occur outside of the program, since it is ignored after the first scan. For example, it may appear on the line preceding the first program line.

Date: 8/4/71
Section: 2.7
Page: 3 of 5
Revision:
IBAL MANUAL

Example:

1	10	72
A	LINE 1, 10, 72	
*	BEGIN	
	SAMPLE PROGRAM	
	END	A

Two additional variations on the LINE directive are provided. The standard line format for IBAL will be defined as the format specified by the directive: LINE 1, 10, 72. Therefore, the occurrence of LINE with no numbers following it will be equivalent to LINE 1, 10, 72. Also, LINE FREE will specify free-format.

Date: 8/4/71
Section: 2.7
Page: 4 of 5
Revision:
IBAL MANUAL

2.7.4 Assembler Directive

```
<assembler directive> ::= DC   , [(<unsigned integer>)] <constant>     
| SETCS <cell size>  
| SETNT <number type>  
| SETRV <register variant>  
| FILNUL <cell size>  
| FILNOP <cell size>
```

The assembler directives cause action only at translation time.

The DC, define constant, directive is for the insertion of data into a program. The binary representation of each constant is determined, as stated in Section 2.2.4, and placed into store by the assembler at the location assigned to the position in the program where the directive appears. The optional integer preceding the constant is a duplication factor which specifies the number of times that the entire constant will be placed sequentially in the store.

The SETCS, SETNT, and SETRV directives define a cell size, number type, or register variant respectively for all the basic data transfer or arithmetic instructions within their scope which do not have the optional field designator or number type specified. The scope of a SETxx directive is all instructions following the SETxx directive up to the next SETxx directive.

The FILNUL or FILNOP directives position the object code pointer on a boundary which is a multiple of the specified cell size by filling the next n bytes (where n < cell size) with zeroes or NOP instructions respectively, and incrementing the object code pointer as each byte is filled.

Date: 8/4/71
Section: 2.7
Page: 5 of 5
Revision:
IBAL MANUAL

3. Segmentation

An important aspect of the storage allocation performed by the IBAL translator is segmentation. Segmentation is desirable when operating in a time-sharing or multi-programming environment. A segment is a group of consecutively-addressed words in storage with a fixed maximum length. Segments are the basic units handled in allocation between primary and secondary storage devices. During the execution of a block B, only those segments whose scope includes B will need to be available in primary storage. ,

3.1 Procedure Segments

A program will be segmented according to block structure. The user may specify that a procedure block is to be a separate segment by including the reserved word `SEGMENT` after the reserved word `PROCEDURE` at the beginning of the block. Compare Wirth [6].

In addition, the translator will segment a program even when the user does not specify it, if the program is too large. The user will be informed of the action taken. All the executable instructions of one block will always be contained in the same segment. Consequently no program will be compiled if the storage required by one block exceeds the maximum segment size.

Date: 2/9/68
Section: 3.1
Page: 1 of 1
Revision:
IBAL MANUAL

3.2 Data Segments

Data sets declared within a block may also be segmented. The collection of data declared by one DECLARE statement will always be contained in the same segment. The user may specify that a new data segment is to be started by including the reserved word SEGMENT after the reserved word DECLARE in the declaration.

Example:

```
A: BEGIN ;  
    DECLARE F1 (N), 2 W ;  
    DECLARE SEGMENT F2 D, J H ;  
    .  
    .  
    .  
    END A
```

In this example F2 and J will be in the same segment, but F1 will be located in another segment.

Date: 2/9/68
Section: 3.2
Page: 1 of 1
Revision:
IBAL MANUAL

3.3 Storage Attributes

<storage attribute> ::= RDONLY
 | WRONLY
 | CONTIGUOUS

Storage attributes may be specified for procedures or data segments by additional reserved words following the reserved word SEGMENT.

The access rights to the segment may be restricted to read only or write only by the inclusion of the reserved words RDONLY or WRONLY respectively.

The attribute CONTIGUOUS specifies that the segment must occupy contiguous locations in storage. Thus the Illiac III partitioned storage mode may not be used, and the segment specified by the user may not be further segmented.

Date: 3/7/68
Section: 3.3
Page: 1 of 1
Revision:
IBAL MANUAL

3.4 Intersegment Communication

Each program translated by the IBAL translator will be a task in the system. A task is the smallest piece of work which can be scheduled to run in the system. Every task is run as an independent unit of work but it can invoke other tasks which may run simultaneously in the system. Synchronization between tasks is done with asynchronous information which is specified by the caller to the operating system.

Each task in the system has a segment table which contains all information necessary to run the task successfully. Among this information is an entry for each segment known to the task.

Each segment is assigned an internal number which is its index of entry in the segment table. Generally a shared segment has different internal names within tasks which are communicating with it simultaneously.

The IBAL translator generates a segment table with an entry for all internal segments which are known to the program at translation time.

Internal segments are generated by user (specifying SEGMENT attribute) or by translator at compile time. There can be Data, BEGIN, or Procedure Segments. A procedure segment is made of three distinguished parts:

- 1) External reference list: placed at the beginning of the segment
- 2) Prologue: placed before Procedure Code
- 3) Code

All internal segments are made known to the program at the compilation time (but not necessarily all loaded into core memory) and external segments are only made known when they are referenced at run time. For details of dynamic intersegment linking, see Illiac III Reference Manual, Vol. IV, Supervisor Organization.

4. Pointer Register Allocation

When a pointer is transformed into an operator or operand pointer specification, the tag is initially unknown and the assembler must therefore assign a pointer register to be used. Before execution of the instruction which contains the pointer, the correct pointer value is loaded into the assigned pointer register and the pointer register base name is set to correspond with the base field of the data file. Depending on the complexity of the data file and pointer, the transformation from pointer to primitive phrase may produce a long execution-time access routine.

Pointer register allocation is performed by the IBAL system in the following manner:

1. The number of base fields used by the program data items is determined.
2. Pointer registers 7 through 12 are allocated in a cyclic order to the base fields.
3. Pointer registers assigned to one base field are assigned in a cyclic order to pointers of this base field which appear in the program.
4. If a control operation intervenes between uses of the same pointer, the pointer value will be reloaded into the allocated register.

Thus, the file system uses pointer registers 7 through 12. If the user has pointers to data elements in his program, he must be careful about the use of these registers.

Example:

Program

```
P:      BEGIN                ;  
        DECLARE A W, B W, C W ;  
        SETCS   W            ;
```

Date: 2/9/68
Section: 4
Page: 1 of 2
Revision:
IBAL MANUAL

```

        PUSH      , A          ;      8
        PUSH      , C          ;     11
        .
        .
        .
S:      BEGIN                ;
        DECLARE D W, E W, F W ;
        .
        .
        .
        PUSH      , D          ;      9
        PUSH      , B          ;      8
        TEST      , D          ;      9
        POP       , F          ;     12
        .
        .
        .
        END       S           ;
        END       P           ;

```

If A, B, C are assigned to base field 1, and D, E, F are assigned to base field 2, and if the following assignments of pointer registers to these base fields is made:

```

BR1      -      PR8          ,      PR11
BR2      -      PR9          ,      PR12

```

then the pointer register assignment to pointers is shown in the right hand column.

Date: 2/9/68
 Section: 4
 Page: 2 of 2
 Revision:
 IBAL MANUAL

BIBLIOGRAPHY

1. McCormick, B. H., Nordmann, B. J., Jr., (ed.) "Illiac III Reference Manual, Volume I: The Computer System, Department of Computer Science Report No. 433, February 17, 1971.
2. McCormick, B. H., Nordmann, B. J., Jr., (ed.) "Illiac III Reference Manual, Volume II: Instruction Repertoire", Department of Computer Science Report No. 434, February 26, 1971.
3. McCormick, B. H., Nordmann, B. J., Jr., (ed.) "Illiac III Reference Manual, Volume III: Input/Output", Department of Computer Science Report (in progress).
4. McCormick, B. H., Nordmann, B. J., Jr., (ed.) "Illiac III Reference Manual, Volume IV: Supervisor Organization", Department of Computer Science Report (in progress).
5. Naur, P. (ed.), Revised Report on the Algorithmic Language ALGOL 60. Communications of the ACM 6 (Jan. 1963), 1-17.
6. IBM System/360 Operating System, PL/1 (F), Language Reference Manual, IBM Systems Reference Library, File NO. S360-29, Form C28-8201-2.
7. Wilbur, J. A., "A Language for Describing Digital Computers", Department of Computer Science, University of Illinois, Report No. 197, February, 1966.
8. Bottenbruch, H., "Structure and Use of ALGOL 60." Journal of the ACM 9 (April 1962), 161-221.
9. Wirth, N., "A Programming Language for the 360 Computers." Journal of the ACM 15 (Jan. 1968), 37-74.

Date: 7/26/71
Section: Bibliography
Page: 1 of 1
Revision:
IBAL MANUAL

APPENDIX 1. THE 60 CHARACTER SET

A2.1 Alphabetic Characters

Letters	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
Currency Symbol	\$
Commercial At-sign	@
Number Sign	#

A2.2 Numeric Characters

Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
--------	------------------------------

A2.3 Special Characters

Blank	
Equal or Assignment	=
Plus	+
Minus	-
Asterisk or Multiply	*
Slash or Divide	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point or Period	.
Quotation Mark	'
Percent	%
Semicolon	;
Colon	:
Not	¬
And	&
Or	
Greater Than	>

Date: 2/9/68
Section: Appendix 1
Page: 1 of 2
Revision:
IBAL MANUAL

Less Than	<
Break Character (Used as shown)	-
Question Mark	?

Date: 2/9/68
Section: Appendix 1
Page: 2 of 2
Revision:
IBAL MANUAL

Appendix 2. Reserved Words

ABS ACTP ADD ALLOCATE AND AREA ASSIGN
B BEGIN BIT BQBLE
C CALL CLEARP CONNECT CONTIGUOUS CONTROL COPY COPYC COUNT CPRA CPRL CS CVD CVF CVL
D DC DECL DECLARE DECR DIV DUP
E EDIT ELSE END ENTRY EQU EQV ERASEP EVENT EXECUTE EXIT
F FILNOP FILNUL FM FREE
GATEIA GETC GQ GQTQ
H HEXAGONAL HIQ
IF IFN INCK INCL INCR INITIAL INRT IAWORD
L LAR LBR LD LIBR LIKE LINE LINK LIST LISTI LISTLZ LISTSZ LQAD LQADB LQC LS LTR
MNS MODIFY MOVE MOVEB MULT
N NE NW NEG NOP NOT
ONE OR OS OV
P PACK PLANAR PLAND PLEQV PLEXOR PLNAND PLNOR PLOR PLOT PLOTI PLOTLZ PLOTSZ
POLY POP POPB POPF POPFR PRO PR1 PR2 PR3 PR4 PR5 PR6 PR7 PR8 PR9 PR10 PR11
PR12 PR13 PR14 PRIORITY PROCEDURE PUSH PUSHB PUSHF PUSHFR PUTC
R RDCLK RDERLZ RDONLY RDTIM READ READLZ READR RECTANGULAR RENA REPLICATE
RESET RESETM RESTART RESU RESUME RETURN RS
S SAR SBR SCAN SCANM SE SEGMENT SENSE SET SETCS SETM SETNT SETORG SETP SETRV
SHIFT SIBR SIQ SIM SL SLEEP SLUFF SPECIFY SR ST STOP STORE STOREB STR STRING
STRUCTURE STTIM SUB SVC SVR SW
TA TALLY TALLYH TASK TEST TESTALL TESTANY TESTB TESTDS TESTM TESTP THEN TIQ
TOPLOGY TORIDAL TRANS
UNPACK
V VECTOR
W WAIT WHQ WRERLZ WRITE WRITLZ WRONGLY
X XCH XOR
ZERO

Date: 7/26/71
Section: Appendix 2
Page : 1 of 1
Revision:
IBAL MANUAL

Appendix 3. Complete IBAL Syntax

Basic Elements

<alphabetic> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|\$|@|#

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

<special character> ::= +|-|=|*|/|(.)|,|.|'|%|;|:|~|_?<|>|<|_?<|>|<|_?

<alphanumeric> ::= <alphabetic>|<digit>

<basic character> ::= <alphanumeric>|<special character>

<identifier> ::= <alphabetic> | <alphanumeric> [-] _

<unsigned integer> ::= ||<digit>||

<comment> ::= /*<any basic character string not containing */> */

<name> ::= <identifier>

<event name> ::= <identifier>

<task name> ::= <identifier>

<parameter> ::= <identifier>

<label> ::= <identifier>

<symbol> ::= <identifier>

|<register symbol>

<register symbol> ::= ØS|PRO|PR1|PR2|PR3|PR4|PR5|PR6|PR7|PR8|PR9|PR10|PR11|PR12
|PR13|PR14

<cell size> ::= B|H|W|D

<register variant> ::= F|L|V|R

<number type> ::= S|L|F|D

<hexadigit> ::= <digit> |A|B|C|D|E|F

<bit> ::= 0|1

<sign> ::= +|-

<exponent> ::= E[<sign>] <unsigned integer>

<scale field> ::= S[<sign>] <unsigned integer>

<decimal number> ::= [<sign>] {<unsigned integer> [.] [<unsigned integer>]
|. <unsigned integer>} [<exponent>]

Date: 7/26/71

Section: Appendix 3

Page: 1 of 9

Revision:

IBAL MANUAL

$\langle \text{binary number} \rangle ::= _ _ _ \langle \text{bit} \rangle _ _ _ [\langle \text{scale field} \rangle]$
 $\langle \text{hex number} \rangle ::= _ _ _ \langle \text{hexadigit} \rangle _ _ _ [\langle \text{scale field} \rangle]$
 $\langle \text{character string} \rangle ::= \langle \text{string of basic characters with ' occurring only in adjacent pairs} \rangle$
 $\langle \text{constant data bits} \rangle ::= \langle \text{character string} \rangle ' C$
 $\quad \quad \quad | \langle \text{binary number} \rangle _ , \langle \text{binary number} \rangle _ ' B$
 $\quad \quad \quad | \langle \text{hex number} \rangle _ , \langle \text{hex number} \rangle _ ' X$
 $\quad \quad \quad | \langle \text{decimal number} \rangle \langle \text{number type} \rangle$
 $\langle \text{constant flag bits} \rangle ::= ' _ _ _ \langle \text{bit} \rangle _ _ _ ' F$
 $\langle \text{constant} \rangle ::= \langle \text{constant data bits} \rangle [\langle \text{constant flag bits} \rangle]$
 $\quad \quad \quad | \langle \text{constant flag bits} \rangle$
 $\langle \text{literal} \rangle ::= = \langle \text{constant} \rangle$
 $\langle \text{self-defining value} \rangle ::= \langle \text{unsigned integer} \rangle$
 $\quad \quad \quad | \langle \text{binary number} \rangle ' B$
 $\quad \quad \quad | \langle \text{hex number} \rangle ' X$
 $\quad \quad \quad | \langle \text{character string} \rangle ' C$
 $\langle \text{primary} \rangle ::= \langle \text{self-defining value} \rangle$
 $\quad \quad \quad | \langle \text{symbol} \rangle$
 $\quad \quad \quad | \langle \text{label} \rangle$
 $\quad \quad \quad | (\langle \text{constant expression} \rangle)$
 $\langle \text{term} \rangle ::= [\langle \text{term} \rangle \{ * | / \}] \langle \text{primary} \rangle$
 $\langle \text{constant expression} \rangle ::= [[\langle \text{constant expression} \rangle] \{ + | - \}] \langle \text{term} \rangle$
 $\langle \text{integer expression} \rangle ::= \langle \text{constant expression} \rangle$
 $\quad \quad \quad | \langle \text{operand} \rangle$

Program

$\langle \text{statement} \rangle ::= \langle \text{declaration} \rangle$
 $\quad \quad \quad | \langle \text{instruction} \rangle$
 $\quad \quad \quad | \langle \text{directive} \rangle$
 $\quad \quad \quad | \langle \text{block} \rangle$
 $\quad \quad \quad | \langle \text{label} \rangle : \langle \text{statement} \rangle$

Date: 2/14/68
 Section: Appendix 3
 Page: 2 of 9
 Revision:
 IBAL MANUAL

```

<block> ::= BEGIN ; | <statement>; | END
          | PROCEDURE[ SEGMENT | <storage attribute> | ] [ ( <parameter> | , <parameter> | ) ];
          | <statement>; | END
          | <label> : <block> [ <label> ]

```

```

<program> ::= <block>;

```

```

<storage attribute> ::= RDONLY | WRONLY | CONTIGUOUS

```

Declaration

```

<declaration> ::= { DECLARE [ SEGMENT | <storage attribute> | ] | STRUCTURE }
                 <format item> | { , | * } <format item> |

```

```

<format item> ::= [ <level> ] [ [ * ] <name> ] [ <attributes> ]

```

```

<level> ::= <unsigned integer>

```

```

<attributes> ::= <nonterminal node attributes>
                | <terminal node attribute>

```

```

<nonterminal node attributes> ::= [ ( <bounds> | , <bounds> | ) ]
                                   [ { STRING | VECTOR } ]

```

```

<terminal node attribute> ::= <cell size> | <constant> | LIKE <name>

```

```

<bounds> ::= [ <lower bound> : ] <upper bound>

```

```

    | > <bound>

```

```

    | < <bound>

```

```

    | *

```

```

<bound> ::= <integer expression>

```

```

<lower bound> ::= <integer expression>

```

```

<upper bound> ::= <integer expression>

```

```

<constant list> ::= [ ( <unsigned integer> ) ] <cspec> | [ ( <unsigned integer> ) ] <cspec> |

```

```

<cspec> ::= <constant> | *

```

```

<conditional format item> ::= <if clause> <simple format item>

```

```

                        ELSE <simple format item>

```

```

<if clause> ::= IF <Boolean expression> THEN

```

```

<Boolean expression> ::= <Boolean term> | <Boolean term> |

```

```

<Boolean term> ::= <Boolean factor> | & <Boolean factor> |

```

```

<Boolean factor> ::= ( <Relation> ) | <Logical value>
                    [ 7 ] ( <Boolean expression> )

```

```

<Logical value> ::= FALSE | TRUE

```

```

<relation> ::= <integer expression> <relational operator>

```

```

                <integer expression>

```

```

<relational operator> = > | >= | = | <= | < | <7

```

Date: 8/5/70
Section: Appendix 3
Page: 3 of 9
Revision:
IBAL MANUAL

Instruction

`<instruction> ::= <imprimitive instruction>`
 `| <basic data transfer>`
 `| <logical>`
 `| <string and list processing>`
 `| <segment administration>`
 `| <arithmetic>`
 `| <decision>`
 `| <supervisory>`
 `| <input/output>`
 `| <pattern articulation>`
 `| NOP`

`<modifier> ::= <constant expression>`
`<modification operator> ::= + | - | = | *`
`<tag> ::= <symbol> | <self-defining value>`
`<pointer specification> ::= <tag> [ * ]`
 `| <tag> <modification operator> <modifier>`
 `| <pointer>`
`<operand> ::= [ / ] <pointer specification> [ / ]`
 `| <literal>`
`<entry name> ::= <label>`
`<operator mnemonic> ::= GØ TØ | CALL | EXECUTE`
`<operator> ::= <operator mnemonic> <pointer specification>`
 `| GØ TØ <label>`
 `| CALL <entry name> <asynchronous information>`
 `| EXECUTE <label>`
 `| SPECIFY`

`<imprimitive instruction> ::= <operator>   , <operand>    | EXIT`
`<subscript> ::= <integer expression>`
`<pointer> ::= <name> [ ( <subscript>   , <subscript>    ) ]`
 `. <name> [ ( <subscript>   , <subscript>    ) ]`

Date: 3/7/68
Section: Appendix 3
Page: 4 of 9
Revision:
IBAL MANUAL

<asynchronous information> ::= [TASK [(<task name>)]]
 | EVENT (<event name>)
 | PRIORITY (<integer expression>)]

<field designator> ::= <cell size> | <register variant>

<basic data transfer> ::= <type 1> [<field designator>], [...] <operand>
 | <type 2> [<field designator>]
 , [...] <operand>, [...] <operand>

<type 1> ::= PUSH|POP|LD|ST|SET|RESET|TEST|TESTM

<type 2> ::= ASSIGN

<count> ::= <operand>

<logical> ::= <type 3> [<cell size>]
 | <type 4> [<cell size>], <count>
 | <type 5> [<cell size>] [, <count>], <operand>

<type 3> ::= ONE|ZERO|NOT|COUNT|BIT|AND|OR|XOR|EQV|CPRL|XCH|DUP|SLUFF

<type 4> ::= LS|RS

<type 5> ::= SCAN|SCANM

<string and list processing> ::= <type 6> [, <count>], <operand>
 | <type 7>, <operand>
 | <type 8> [, <count>], <operand>, <operand>

<type 6> ::= PUSHF|POPF|PUSHFR|POPFR|PACK|UNPACK

<type 7> ::= SL|SR|GETC|PUTC|INCL|INCR|DECL|DECR

<type 8> ::= EDIT|TRANS|MOVE

<segment administration> ::= <type 9>, <operand>, <operand>
 | <type 10>

<type 9> ::= LBR|SBR

<type 10> ::= LAR|SAR|LOC

<arithmetic> ::= <type 11> [<number type>]

<type 11> ::= NEG|ABS|MNS|TA|ADD|SUB|MULT|DIV|CPRA|POLY|CVD|CVF|CVL

<decision> ::= <type 12> 1, <condition> 1, <label>

<type 12> ::= IF|IFN

<condition> ::= <|=|>|FM|OV|CS

Date: 2/14/68 Section: Appendix 3 Page: 5 of 9 Revision: IBAL MANUAL
--

< system> ::= < supervisory> | < interrupt> | < input-output> |
 < coordinating> | < timing>

< supervisory> ::= { SVC | SVR | ACTP | LTR | STR }
 | SLEEP, < operand>
 | { RNAM | RESU }, < operand>, < operand>

< interrupt> ::= INRT | SIM, < operand>

< input-output> ::= { SIO | HIO | LIBR }, < operand>

< coordinating> ::= WHO | INCK, < operand>

< timing> ::= RDCLK | STTIM | RDTIM


```

< command> ::= < interrupt command> | < device dependent command>
               | < device independent command>

< interrupt command> ::= SIO, < channel name>, < pointer value>
               | , < channel name> |
               | HIO, < Program tag>
               | LIBR, < BR tag>, < base register contents>

< channel name> ::= < constant expression>

< pointer value> ::= < constant expression>

< program tag> ::= < symbol> | < self defining value>

< BR tag> ::= < symbol> | < self defining value>

< base register contents> ::= < constant expression>

< device dependent command> ::= < DDC mnemonic> | < DDC tag> | ,
               < extended device address>

< DDC mnemonic> ::= READ | READR | SENSE | WRITE | CONTROL | TESTDS | POLL

< DDC tag> ::= RD | DEI | P | PO | R

< extended device address> ::= < rate code>, < channel name>,
               < device address>

< rate code> ::= < constant expression>

< device address> ::= < constant expression>

< device independent command> ::= < DIC mnemonic> < field designator>,
               < DIC tag> < modification operator>
               < modifier>

< DIC mnemonic> ::= LOAD | STORE | SETM | RESETM | TESTANY | TESTALL | MODIFY | STOP

< DIC tag> ::= < program or channel>, < name>, < register address>

< program or channel> ::= P | C

< name> ::= < identifier> | < self defining value>

< register address> ::= < identifier> | < self defining value>

```

```

<pattern articulation> ::= <0-plane>
                                | <1-plane>
                                | <2-plane>
                                | <3-plane>
                                | <multi-plane>
                                | <border>

0-plane ::= TOPOLOGY [ {RECTANGULAR|HEXAGONAL} ] [ {PLANAR|TOROIDAL} ] RESTART
           | RESUME | SETORG, <operand>

<1-plane> ::= {CLEARP|SETP}, l, <ID variant> l, <plane name>
              | {TESTP|TESTB|LIST|ERASEP}, <plane name>
              | SHIFT, <plane name>, <displacement>
              | {TALLY|TALLYHØ}, <plane name>, <direction list>
              | {AREA|LIST|LISTSZ|LISTLZ}, <plane name> [, <count> ]
              | {PLOT|PLOTSZ|PLOTLZ|PLOTI}, <plane name>, <string name>
              | READLZ|RDERLZ, <plane name>
              | {WRITLZ|WRERLZ}, <plane name>

<ID variant> ::= P|E|W|N|S|NE|NW|SE|SW

<plane name>  := <operand>

<displacement> ::= <operand>

<direction list> ::= <operand>

<string name> ::= <operand>

<2-plane> ::= {COPY|COPYC |PLAND|PLØR|PLNAND|PLNØR|PLEXØR|PLEQV}

<3-plane> ::= CØNNECT, <plane name>, <plane name> [, <plane name> ], <direction list>

<multi-plane> ::= REPLICATE
                 | BØØLE, <availability list>, <operand> | GATEIA, <IAWORD word>

<availability list> ::= <constant> | <symbol>

<IAWORD word> ::= <constant> | <symbol>

<border> ::= {LOADB|STOREB|PUSHB|POPb}, <gbw>, <string name>
             | MØVEB, <gbw>, <plane name>, <plane name>

<gbw> ::= <constant> | <symbol>

```

Date: 7/26/71 Section: Appendix 3 Page: 8 of 9 Revision: IBAL MANUAL
--

Directive

<directive> ::= <control directive>

| <assignment directive>
| <source format directive>
| <assembler directive>

<control directive> ::= ENTRY

| RETURN
| WAIT <event name>

<assignment directive> ::= ALLØCATE||, <pointer> -> <tag>||
| EQU , <constant expression>

<column> ::= <unsigned integer>

<source format directive> ::= LINE[<column>,<column>,<column>]
| LINE FREE

<assembler directive> ::= DC ||,[(<unsigned integer>)]<constant>||
| SETCS<cell size>
| SETNT<number type>
| SETRV<register variant>
| FILNUL<cell size>
| FILNØP<cell size>

Date: 3/8/68
Section: Appendix 3
Page: 9 of 9
Revision:
IBAL MANUAL

Appendix 4. Sample Program Segment

/* Sample IBAL Program Segment */

PROG: BEGIN;

STRUCTURE 1 V(10) , 2 W ;

DECLARE A LIKE V ,

B LIKE V ,)

C LIKE V ;

CALL VECTORADD, A(1), B(1), C(1), PR6 = 10;

VECTORADD: PROCEDURE (X, Y, Z, CNT);

/* PERFORMS ADDITION $Z = X + Y$ WHERE

Z, X, Y are vectors of CNT elements */

SPECIFY , X - 4, Y - 4, Z - 4 ;

LOOP: PUSH W , X + 4 ;

PUSH W , Y + 4 ;

ADD L ;

POP W , Z + 4 ;

GO TO LOOP , CNT - 1 ;

EXIT ;

END VECTORADD ;

END PROG ;

Date: 3/8/68
Section: Appendix 4
Page: 1 of 1
Revision:
IBAL MANUAL

Zero Operand Instruction Codes -

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	ZERO (B)	1 0 0 0 0 0	SLUFF (B)
0 0 0 0 0 1	ZERO (H)	1 0 0 0 0 1	SLUFF (H)
0 0 0 0 1 0	ZERO (W)	1 0 0 0 1 0	SLUFF (W)
0 0 0 0 1 1	ZERO (D)	1 0 0 0 1 1	SLUFF (D)
0 0 0 1 0 0	ONE (B)	1 0 0 1 0 0	DUP (B)
0 0 0 1 0 1	ONE (H)	1 0 0 1 0 1	DUP (H)
0 0 0 1 1 0	ONE (W)	1 0 0 1 1 0	DUP (W)
0 0 0 1 1 1	ONE (D)	1 0 0 1 1 1	DUP (D)
0 0 1 0 0 0	COUNT (B)	1 0 1 0 0 0	XCH (B)
0 0 1 0 0 1	COUNT (H)	1 0 1 0 0 1	XCH (H)
0 0 1 0 1 0	COUNT (W)	1 0 1 0 1 0	XCH (W)
0 0 1 0 1 1	COUNT (D)	1 0 1 0 1 1	XCH (D)
0 0 1 1 0 0	BIT (B)	1 0 1 1 0 0	CPRL (B)
0 0 1 1 0 1	BIT (H)	1 0 1 1 0 1	CPRL (H)
0 0 1 1 1 0	BIT (W)	1 0 1 1 1 0	CPRL (W)
0 0 1 1 1 1	BIT (D)	1 0 1 1 1 1	CPRL (D)
0 1 0 0 0 0	AND (B)	1 1 0 0 0 0	NOT (B)
0 1 0 0 0 1	AND (H)	1 1 0 0 0 1	NOT (H)
0 1 0 0 1 0	AND (W)	1 1 0 0 1 0	NOT (W)
0 1 0 0 1 1	AND (D)	1 1 0 0 1 1	NOT (D)
0 1 0 1 0 0	OR (B)	1 1 0 1 0 0	ACTP (#0)
0 1 0 1 0 1	OR (H)	1 1 0 1 0 1	ACTP (#1)
0 1 0 1 1 0	OR (W)	1 1 0 1 1 0	ACTP (#2)
0 1 0 1 1 1	OR (D)	1 1 0 1 1 1	ACTP (#3)
0 1 1 0 0 0	XOR (B)	1 1 1 0 0 0	INRT
0 1 1 0 0 1	XOR (H)	1 1 1 0 0 1	
0 1 1 0 1 0	XOR (W)	1 1 1 0 1 0	LTR
0 1 1 0 1 1	XOR (D)	1 1 1 0 1 1	STR
0 1 1 1 0 0	EQV (B)	1 1 1 1 0 0	SVC
0 1 1 1 0 1	EQV (H)	1 1 1 1 0 1	SVR
0 1 1 1 1 0	EQV (W)	1 1 1 1 1 0	OKOFF
0 1 1 1 1 1	EQV (D)	1 1 1 1 1 1	WHO

ONE OPERAND INSTRUCTION CODES -

Flag

MN 1

MN 2

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	PUSH (B)	1 0 0 0 0 0	LS (B)
0 0 0 0 0 1	PUSH (H)	1 0 0 0 0 1	LS (H)
0 0 0 0 1 0	PUSH (W)	1 0 0 0 1 0	LS (W)
0 0 0 0 1 1	PUSH (D)	1 0 0 0 1 1	
0 0 0 1 0 0	LD (B)	1 0 0 1 0 0	RS (B)
0 0 0 1 0 1	LD (H)	1 0 0 1 0 1	RS (H)
0 0 0 1 1 0	LD (W)	1 0 0 1 1 0	RS (W)
0 0 0 1 1 1	LD (D)	1 0 0 1 1 1	
0 0 1 0 0 0	POP (B)	1 0 1 0 0 0	
0 0 1 0 0 1	POP (H)	1 0 1 0 0 1	
0 0 1 0 1 0	POP (W)	1 0 1 0 1 0	
0 0 1 0 1 1	POP (D)	1 0 1 0 1 1	
0 0 1 1 0 0	ST (B)	1 0 1 1 0 0	
0 0 1 1 0 1	ST (H)	1 0 1 1 0 1	
0 0 1 1 1 0	ST (W)	1 0 1 1 1 0	
0 0 1 1 1 1	ST (D)	1 0 1 1 1 1	
0 1 0 0 0 0	SET (B)	1 1 0 0 0 0	
0 1 0 0 0 1	SET (H)	1 1 0 0 0 1	
0 1 0 0 1 0	SET (W)	1 1 0 0 1 0	
0 1 0 0 1 1	SET (D)	1 1 0 0 1 1	
0 1 0 1 0 0	RESET (B)	1 1 0 1 0 0	
0 1 0 1 0 1	RESET (H)	1 1 0 1 0 1	
0 1 0 1 1 0	RESET (W)	1 1 0 1 1 0	
0 1 0 1 1 1	RESET (D)	1 1 0 1 1 1	
0 1 1 0 0 0	TEST (B)	1 1 1 0 0 0	
0 1 1 0 0 1	TEST (H)	1 1 1 0 0 1	
0 1 1 0 1 0	TEST (W)	1 1 1 0 1 0	
0 1 1 0 1 1	TEST (D)	1 1 1 0 1 1	
0 1 1 1 0 0	TESTM (B)	1 1 1 1 0 0	SLEEP
0 1 1 1 0 1	TESTM (H)	1 1 1 1 0 1	INCK
0 1 1 1 1 0	TESTM (W)	1 1 1 1 1 0	SIM
0 1 1 1 1 1	TESTM (D)	1 1 1 1 1 1

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0		1 0 0 0 0 0	
0 0 0 0 0 1		1 0 0 0 0 1	
0 0 0 0 1 0		1 0 0 0 1 0	
0 0 0 0 1 1		1 0 0 0 1 1	
0 0 0 1 0 0		1 0 0 1 0 0	
0 0 0 1 0 1		1 0 0 1 0 1	
0 0 0 1 1 0		1 0 0 1 1 0	
0 0 0 1 1 1		1 0 0 1 1 1	
0 0 1 0 0 0		1 0 1 0 0 0	
0 0 1 0 0 1		1 0 1 0 0 1	
0 0 1 0 1 0		1 0 1 0 1 0	
0 0 1 0 1 1		1 0 1 0 1 1	
0 0 1 1 0 0		1 0 1 1 0 0	
0 0 1 1 0 1		1 0 1 1 0 1	
0 0 1 1 1 0		1 0 1 1 1 0	
0 0 1 1 1 1		1 0 1 1 1 1	
0 1 0 0 0 0		1 1 0 0 0 0	
0 1 0 0 0 1		1 1 0 0 0 1	
0 1 0 0 1 0		1 1 0 0 1 0	
0 1 0 0 1 1		1 1 0 0 1 1	
0 1 0 1 0 0		1 1 0 1 0 0	
0 1 0 1 0 1		1 1 0 1 0 1	
0 1 0 1 1 0		1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0		1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

Appendix 5

con't Instruction Mnemonic Codes

7/26/71

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0		1 0 0 0 0 0	
0 0 0 0 0 1		1 0 0 0 0 1	
0 0 0 0 1 0		1 0 0 0 1 0	
0 0 0 0 1 1		1 0 0 0 1 1	
0 0 0 1 0 0		1 0 0 1 0 0	
0 0 0 1 0 1		1 0 0 1 0 1	
0 0 0 1 1 0		1 0 0 1 1 0	
0 0 0 1 1 1		1 0 0 1 1 1	
0 0 1 0 0 0		1 0 1 0 0 0	
0 0 1 0 0 1		1 0 1 0 0 1	
0 0 1 0 1 0		1 0 1 0 1 0	
0 0 1 0 1 1		1 0 1 0 1 1	
0 0 1 1 0 0		1 0 1 1 0 0	
0 0 1 1 0 1		1 0 1 1 0 1	
0 0 1 1 1 0		1 0 1 1 1 0	
0 0 1 1 1 1		1 0 1 1 1 1	
0 1 0 0 0 0		1 1 0 0 0 0	
0 1 0 0 0 1		1 1 0 0 0 1	
0 1 0 0 1 0		1 1 0 0 1 0	
0 1 0 0 1 1		1 1 0 0 1 1	
0 1 0 1 0 0		1 1 0 1 0 0	
0 1 0 1 0 1		1 1 0 1 0 1	
0 1 0 1 1 0		1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0		1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

7/26/71

Appendix 5 cont. Instruction Mnemonic Codes

Mnemonic Bit: 3 4 5 6 7 8		Instruction	
0 0 0 0 0 0	ASSIGN	0 0 0 1 0 0	POP (B)
0 0 0 0 0 1	ASSIGN	0 0 0 1 0 1	POP (H)
0 0 0 0 1 0	ASSIGN	0 0 0 1 1 0	POP (W)
0 0 0 0 1 1	ASSIGN	0 0 0 1 1 1	POP (D)
0 0 1 0 0 0	POP (B)	0 0 1 0 0 0	POP (B)
0 0 1 0 0 1	POP (H)	0 0 1 0 0 1	POP (H)
0 0 1 0 1 0	POP (W)	0 0 1 0 1 0	POP (W)
0 0 1 0 1 1	POP (D)	0 0 1 0 1 1	POP (D)
0 0 1 1 0 0	PACK	1 0 1 1 0 0	SCAN (B)
0 0 1 1 0 1	UNPACK	1 0 1 1 0 1	SCAN (H)
0 0 1 1 1 0	RNAM	1 0 1 1 1 0	SCAN (W)
0 0 1 1 1 1	LINK	1 0 1 1 1 1	SCAN (W)
0 1 0 0 0 0	CALL	1 1 0 0 0 0	MOVE
0 1 0 0 0 1	EXECUTE	1 1 0 0 0 1	TRANS
0 1 0 0 1 0	GOTO	1 1 0 0 1 0	EDIT
0 1 0 0 1 1	EXIT	1 1 0 0 1 1	IF
0 1 0 1 0 0	LOC	1 1 0 1 0 0	IF
0 1 0 1 0 1	SPECIFY	1 1 0 1 0 1	IF
0 1 0 1 1 0	NOP	1 1 0 1 1 0	IF
0 1 0 1 1 1	RESU	1 1 0 1 1 1	IF
0 1 1 0 0 0		1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0		1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

Two operands

MISC. INSTRUCTION CODES -

ZERO AND ONE OPERAND INSTRUCTION CODES -

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	LIBR	1 0 0 0 0 0	RDCLK
0 0 0 0 0 1	SL	1 0 0 0 0 1	
0 0 0 0 1 0	SIO	1 0 0 0 1 0	STTIM
0 0 0 0 1 1	HIO	1 0 0 0 1 1	RDTIM
0 0 0 1 0 0	SR	1 0 0 1 0 0	
0 0 0 1 0 1		1 0 0 1 0 1	
0 0 0 1 1 0		1 0 0 1 1 0	
0 0 0 1 1 1		1 0 0 1 1 1	
0 0 1 0 0 0	GET	1 0 1 0 0 0	
0 0 1 0 0 1		1 0 1 0 0 1	
0 0 1 0 1 0		1 0 1 0 1 0	
0 0 1 0 1 1		1 0 1 0 1 1	
0 0 1 1 0 0	PUT	1 0 1 1 0 0	
0 0 1 1 0 1		1 0 1 1 0 1	
0 0 1 1 1 0		1 0 1 1 1 0	
0 0 1 1 1 1		1 0 1 1 1 1	
0 1 0 0 0 0	INCL	1 1 0 0 0 0	
0 1 0 0 0 1		1 1 0 0 0 1	
0 1 0 0 1 0		1 1 0 0 1 0	
0 1 0 0 1 1		1 1 0 0 1 1	
0 1 0 1 0 0	INCR	1 1 0 1 0 0	
0 1 0 1 0 1		1 1 0 1 0 1	
0 1 0 1 1 0		1 1 0 1 1 0	
0 1 0 1 1 1		1 1 0 1 1 1	
0 1 1 0 0 0	DECL	1 1 1 0 0 0	
0 1 1 0 0 1		1 1 1 0 0 1	
0 1 1 0 1 0		1 1 1 0 1 0	
0 1 1 0 1 1		1 1 1 0 1 1	
0 1 1 1 0 0	DECR	1 1 1 1 0 0	
0 1 1 1 0 1		1 1 1 1 0 1	
0 1 1 1 1 0		1 1 1 1 1 0	
0 1 1 1 1 1		1 1 1 1 1 1	

← one operand → ← zero operand →

ARITHMETIC OPERATION CODES -

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0		1 0 0 0 0 0	ADD (SFX)
0 0 0 0 0 1		1 0 0 0 0 1	ADD (LFX)
0 0 0 0 1 0		1 0 0 0 1 0	ADD (FLT)
0 0 0 0 1 1		1 0 0 0 1 1	ADD (DEC)
0 0 0 1 0 0	CVL (SFX)	1 0 0 1 0 0	
0 0 0 1 0 1	NOP	1 0 0 1 0 1	
0 0 0 1 1 0	CVL (FLT)	1 0 0 1 1 0	
0 0 0 1 1 1	CVL (DEC)	1 0 0 1 1 1	
0 0 1 0 0 0	CVF (SFX)	1 0 1 0 0 0	SUB (SFX)
0 0 1 0 0 1	CVF (LFX)	1 0 1 0 0 1	SUB (LFX)
0 0 1 0 1 0	NOP	1 0 1 0 1 0	SUB (FLT)
0 0 1 0 1 1	CVF (DEC)	1 0 1 0 1 1	SUB (DEC)
0 0 1 1 0 0	CVD (SFX)	1 0 1 1 0 0	CPRA (SFX)
0 0 1 1 0 1	CVD (LFX)	1 0 1 1 0 1	CPRA (LFX)
0 0 1 1 1 0	CVD (FLT)	1 0 1 1 1 0	CPRA (FLT)
0 0 1 1 1 1	NOP	1 0 1 1 1 1	CPRA (DEC)
0 1 0 0 0 0	NEG (SFX)	1 1 0 0 0 0	MPY (SFX)
0 1 0 0 0 1	NEG (LFX)	1 1 0 0 0 1	MPY (LFX)
0 1 0 0 1 0	NEG (FLT)	1 1 0 0 1 0	MPY (FLT)
0 1 0 0 1 1	NEG (DEC)	1 1 0 0 1 1	MPY (DEC)
0 1 0 1 0 0	ABS (SFX)	1 1 0 1 0 0	POLY (SFX)
0 1 0 1 0 1	ABS (LFX)	1 1 0 1 0 1	POLY (LFX)
0 1 0 1 1 0	ABS (FLT)	1 1 0 1 1 0	POLY (FLT)
0 1 0 1 1 1	ABS (DEC)	1 1 0 1 1 1	POLY (DEC)
0 1 1 0 0 0	MNS (SFX)	1 1 1 0 0 0	DIV (SFX)
0 1 1 0 0 1	MNS (LFX)	1 1 1 0 0 1	DIV (LFX)
0 1 1 0 1 0	MNS (FLT)	1 1 1 0 1 0	DIV (FLT)
0 1 1 0 1 1	MNS (DEC)	1 1 1 0 1 1	DIV (DEC)
0 1 1 1 0 0	TA (SFX)	1 1 1 1 0 0	
0 1 1 1 0 1	TA (LFX)	1 1 1 1 0 1	
0 1 1 1 1 0	TA (FLT)	1 1 1 1 1 0	
0 1 1 1 1 1	TA (DEC)	1 1 1 1 1 1	

Appendix 5 continued Instruction Mnemonic Codes

7/26/71

PAU INSTRUCTION CODES

MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION	MNEMONIC BIT: 3 4 5 6 7 8	INSTRUCTION
0 0 0 0 0 0	RESTART	1 0 0 0 0 0	AREA
0 0 0 0 0 1	RESUME	1 0 0 0 0 1	LIST
0 0 0 0 1 0	BOOLE	1 0 0 0 1 0	LISTSZ
0 0 0 0 1 1	SETORG	1 0 0 0 1 1	LISTLZ
0 0 0 1 0 0		1 0 0 1 0 0	REPLICATE
0 0 0 1 0 1		1 0 0 1 0 1	RDERLZ
0 0 0 1 1 0		1 0 0 1 1 0	WRITLZ
0 0 0 1 1 1		1 0 0 1 1 1	WRERLZ
0 0 1 0 0 0	GATEIA	1 0 1 0 0 0	TESTP
0 0 1 0 0 1		1 0 1 0 0 1	TESTB
0 0 1 0 1 0		1 0 1 0 1 0	LISTI
0 0 1 0 1 1		1 0 1 0 1 1	ERASEP
0 0 1 1 0 0		1 0 1 1 0 0	SHIFT
0 0 1 1 0 1		1 0 1 1 0 1	TALLY
0 0 1 1 1 0		1 0 1 1 1 0	TALLYHO
0 0 1 1 1 1		1 0 1 1 1 1	CONNECT
0 1 0 0 0 0	LOADB	1 1 0 0 0 0	CLEARP
0 1 0 0 0 1	PUSHB	1 1 0 0 0 1	SETP
0 1 0 0 1 0	STOREB	1 1 0 0 1 0	MOVEB
0 1 0 0 1 1	POPB	1 1 0 0 1 1	READLZ
0 1 0 1 0 0		1 1 0 1 0 0	PLOT
0 1 0 1 0 1		1 1 0 1 0 1	PLOTSZ
0 1 0 1 1 0		1 1 0 1 1 0	PLOTLZ
0 1 0 1 1 1		1 1 0 1 1 1	PLOTI
0 1 1 0 0 0		1 1 1 0 0 0	COPY
0 1 1 0 0 1		1 1 1 0 0 1	COPYC
0 1 1 0 1 0		1 1 1 0 1 0	PLAND
0 1 1 0 1 1		1 1 1 0 1 1	PLOR
0 1 1 1 0 0	TOPOLOGY	1 1 1 1 0 0	PLVAND
0 1 1 1 0 1	TOPOLOGY	1 1 1 1 0 1	PLNOR
0 1 1 1 1 0	TOPOLOGY	1 1 1 1 1 0	PLEXOR
0 1 1 1 1 1	TOPOLOGY	1 1 1 1 1 1	PLEQV