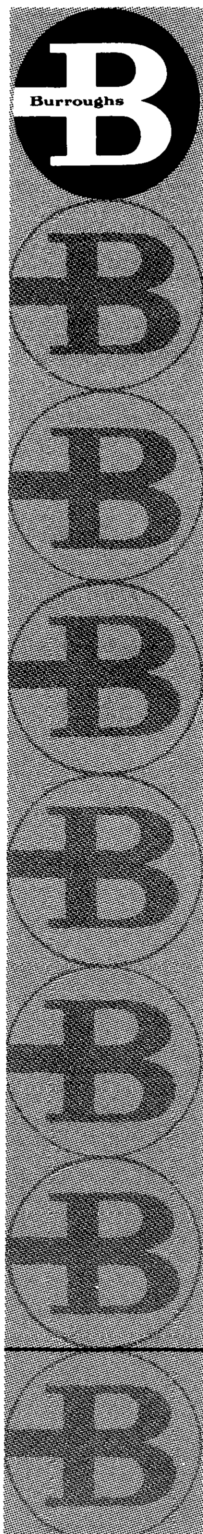
66072

IL4-PM2
17 MARCH 1969



REFERENCE MANUAL

FOR

ILLIAC IV ASSEMBLER (ASK)

Burroughs Corporation
Defense, Space and Special Systems Group

PREFACE

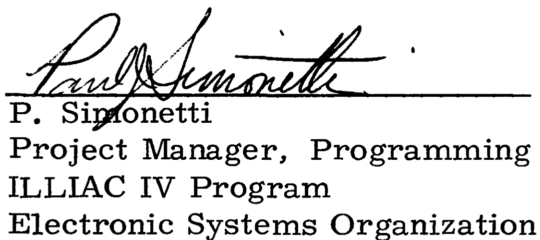
This document provides a description of the basic ILLIAC IV assembly language, ASK version II. Included are:

- A syntactic description of an ASK program,
- A definition of assembly-time arithmetic,
- A definition of assembly-time functions (pseudo operations),
- A definition of assembly-time utilities (control cards), and
- Examples of system control cards necessary to use ASK.

The document assumes some prior familiarity with ASK and a complete familiarity with the instruction set for ILLIAC IV as defined in the "ILLIAC IV Systems Characteristics and Programming Manual."

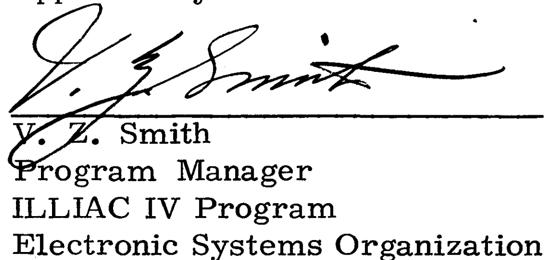
This program was written by Mr. David Grothe of the University of Illinois. This manual was prepared by Mr. Grothe and Mrs. Charlene Luskin, the latter a member of the Special Systems Programming Department.

Submitted by:



P. Simonetti
Project Manager, Programming
ILLIAC IV Program
Electronic Systems Organization

Approved by:



V. Z. Smith
Program Manager
ILLIAC IV Program
Electronic Systems Organization

TABLE OF CONTENTS

NOTE

The various elements of the ILLIAC IV assembly language are discussed in paragraphs labeled Syntax, Semantics, and Restrictions, immediately following each pertinent subject heading. To avoid needless repetition, these subordinate headings have been omitted from the Table of Contents.

Section	Title	Page
I	INTRODUCTION	1-1
	I.A The Assembler	1-1
	I.B Purpose of Manual	1-1
	I.C Card Format	1-1
	I.D Conventions Used in the Description of the Language	1-2
II	ELEMENTS OF THE LANGUAGE	2-1
	II.A Character	2-1
	II.B Symbols	2-1
	II.C Numbers	2-2
III	ARITHMETIC AND ARITHMETIC EXPRESSIONS	3-1
	III.A Arithmetic Expression	3-1
	III.B Arithmetic with Relocatable Quantities	3-2
	III.B.1 Addition	3-3
	III.B.2 Subtraction	3-4
	III.B.3 Multiplication	3-4
	III.B.4 Integer Division	3-5
	III.C Three Modes of Arithmetic: Syllable, Word, and Row	3-6
IV	STRUCTURE OF AN ASK PROGRAM	4-1
V	STATEMENTS	5-1

Section	Title	Page
VI	REGISTER DESIGNATORS AND OPERAND FIELDS FOR CU INSTRUCTIONS	6-1
	VI.A CU Operand Fields	6-1
	VI.B Register Designators in CU	6-5
VII	REGISTER DESIGNATORS AND OPERAND FIELDS FOR PE INSTRUCTIONS	7-1
	VII.A PE Operand Fields	7-1
	VII.B Register Designators in PE	7-4
VIII	OPERAND FIELDS FOR MODE SETTING INSTRUCTIONS	8-1
IX	ASK PSEUDO OPERATIONS	9-1
	IX.A General	9-1
	IX.B EQU Pseudo	9-1
	IX.C SYL Pseudo	9-2
	IX.D WDS Pseudo	9-2
	IX.E BLK Pseudo	9-3
	IX.F FILL Pseudo	9-4
	IX.G SET Pseudo	9-4
	IX.H DATA Pseudo	9-5
	IX.I ORG Pseudo	9-6
	IX.J CHWS Pseudo	9-6
	IX.K REGP Pseudo	9-7
	IX.L REGC Pseudo	9-7
	IX.M SYN Pseudo	9-8
	IX.N GLOBAL Pseudo	9-8
	IX.O LOCAL Pseudo	9-8
	IX.P DEFINE Pseudo	9-9
X	ASK CONTROL STATEMENTS	10-1
XI	PROGRAM DECK	11-1
XII	ERROR MESSAGES	12-1
XIII	INDEX	13-1

I. INTRODUCTION

I.A THE ASSEMBLER

Assembler System K (ASK) is designed to accept a program written in ILLIAC IV assembly language, and to convert it to an ILLIAC IV binary object code.

ASK is a two pass assembler. The following is a brief description of the function of each of the two passes PASS I and PASS II.

In PASS I, ASK determines the values of all symbols defined in the program. It thus performs all pseudo-operations which define symbols. It also checks all instruction mnemonics, and allocate storage as required.

In PASS II, ASK evaluates the operand fields of the ILLIAC IV instructions; hence any ILLIAC IV instruction may reference any defined symbol in its operand field. The ILLIAC IV instructions and data are built and the code is emited to a disk file in a form which is suitable for the ILLIAC IV loader.

I.B PURPOSE OF MANUAL

The purpose of this manual is to supply the programmer with information on how to use ASK and to obtain correct machine code as a result. ASK generates appropriate error messages when incorrect code is encountered. The manual gradually builds the complete syntax acceptable to ASK. The syntax at each level is described by a meta language explained later in this section; the syntax is followed by semantics, restrictions and examples.

I.C CARD FORMAT

An ILLIAC IV assembly source program is either on punched cards or on disk or tape as card images. These cards (or card images) are free form with the following exceptions.

(a) A label must be followed by a colon.

(b) A statement must be followed by a semicolon.

(c) A comment must be preceded by a percent sign. When a % is found on a card ASK does not interpret the remainder of the card.

(d) Columns 73 through 80 are not interpreted by ASK and may contain identification or sequencing information. This field is, however, analyzed when changes are merged with a source tape or disk.

(e) A card containing a $ in column one is recognized as an ASK control statement and specifier certain assembler options.

(f) At least one blank must appear between an instruction and its operand field. With this exception and a few others which are noted in the syntax, blanks may be used freely or omitted without affecting the content of a statement.

(g) Two or more statements (each followed by their required semicolons) may appear on one card.

(h) Identifiers and numbers may not contain embedded blanks or be split across card boundaries.

I.D CONVENTIONS USED IN THE DESCRIPTION OF THE LANGUAGE

The syntax of the language is described through the use of metalinguistic symbols. These symbols have the following meanings:

a. < > Left and right broken brackets are used to contain one or more characters representing a metalinguistic variable whose value is given by a metalinguistic formula.

b. ::= The symbol ::= means "is defined as," and separates the metalinguistic variable on the left of the formula from its definition on the right.

c. | The symbol | means or. This symbol separates multiple definitions of a metalinguistic variable.

d. { } Braces are used to enclose metalinguistic variables which are defined by the meaning of the English-language expression contained within the braces. This formulation is used only when it is impossible or impractical to use a metalinguistic formula.

The above metalinguistic symbols are used in forming a metalinguistic formula. A metalinguistic formula is a rule which will produce an allowable sequence of characters and/or symbols. The entire set of such formulas defines the constructs of ASK (Assembly System K).

Any mark or symbol in a metalinguistic formula which is not one of the above metalinguistic symbols denotes itself. The juxtaposition of metalinguistic variables and/or symbols in a metalinguistic formula denotes justaposition of these elements in the construct indicated.

To illustrate the use of syntax, the following example is offered:

<identifier> ::= <letter> | <identifier> <alphanumeric character>

The above metalinguistic formula is read as follows: an identifier is defined as a letter, or an identifier followed by an alphanumeric character.

The metalinguistic formula defines a recursive relationship by which a construct called an identifier may be formed. Evaluation of the formula shows that an identifier begins with a letter; the letter may stand alone, or may be followed by an mixture of letters and numbers.

II. ELEMENTS OF THE LANGUAGE

II.A CHARACTER

Syntax:

<character> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|Ø|P|Q|R|S|T|U|V|W|X|Y|Z
0|1|2|3|4|5|6|7|8|9|.|[|(|<|←|$|*|)|;|≤|-|/|,|%|=|
]|#| |@|:|>|≥|+|×|≠|?|"

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|Ø|P|Q|R|S|T|U|V|W|X|Y|Z

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

<alphanumeric character> ::= <letter>|<numeric character>

Semantics:

The character set for the assembly language for ILLIAC IV is the 6-bit character set which exists on the Burroughs B5500. An identifier may symbolize things such as a machine instruction, an address in PE memory, or a number. An identifier is restricted to be no more than 63 characters in length.

II.B SYMBOLS

Syntax:

<PE symbol> ::= <identifier>

<CU symbol> ::= .<identifier>

<symbol> ::= <PE symbol>|<CU symbol>

<identifier> ::= <letter>|<identifier> <alphanumeric character>

Semantics:

Although a PE symbol may symbolize an address in PE memory, its semantic interpretation is not restricted to that alone. A PE symbol is best interpreted as symbolizing a number, with the understanding that this number itself takes on quite different meanings depending upon the context in which it is used. ASK (Assembler System K) attaches no meaning (other than its numeric value) to a symbol at the time it is defined.

A PE symbol may have a numeric value of up to 64 bits of precision.

A CU symbol may symbolize an address in CU memory. All the remarks about semantic interpretation of PE symbols apply to CU symbols as well. A CU symbol is restricted to 62 alphanumeric characters in length (+ 1 for the . = 63) and it may assume a value of no greater than

8 bits of precision. If a CU symbol is defined by a quantity of greater precision than 8 bits, the quantity is truncated to 8 bits of precision.

II.C NUMBERS

Syntax:

<integer> ::= <integer part> <base specifier>

<integer part> ::= <base ten digit> | <integer part> <digit>

<base specifier> ::= :<base ten number> | <empty>

<base ten number> ::= <base ten digit> | <base ten number> <base ten digit>

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

<digit> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|G|H|I|J|K|L|M|N|Ø|P|Q|R|S|T|U
V|W|X|Y|Z

<real number> ::= <signed real number> |
<unsigned real number>

<signed real number> ::= +<unsigned real number> |
-<unsigned real number>

<unsigned real number> ::= <base ten number> | <mantissa part> |
<mantissa part> <exponent part> |
<exponent part>

<mantissa part> ::= <base ten number>.|
<base ten number>.<base ten number> | .<base ten number>

<exponent part> ::= @<signed base ten number> |
@<base ten number>

<signed base ten number> ::= +<base ten number> |
-<base ten number>

<paired number> ::= PAIR (<real number or integer>, <real number or integer>)

<real number or integer> ::= <real number> | <integer>

<number> ::= <integer> | <real number> | <paired number>

Semantics:

A number denotes its value. Integers are represented in fixed point binary with the binary point at the right. Real numbers are represented in ILLIAC IV floating point form (see Page 3-3 on data formats ILLIAC IV Systems Characteristics and Programming Manual for details).

A digit must be such that its assigned weight is less than the specified base (or ten if the base is unspecified). The weights assigned to the possible digits are as follows:

digit:	0 1 2 3 4 5 6 7 8 9	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				
weight:	0 1 2 3 4 5 6 7 8 9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
		25	26	27	28	29	30	31	32	33	34	35				

The base specifier directs the assembler to convert the preceding integer from the specified base to binary. If no base is specified, base ten is assumed.

A real number directs the assembler to perform conversion to 64-bit ILLIAC IV floating point representation. This conversion is performed if an explicit decimal point is present or if there is an explicit exponent part. In all other cases, integer conversion is performed.

The pair construct allows for the formation of two 32-bit words in inner-outer form. The first number is converted into the outer position, the second into the inner.

Examples

```
OA:17
31
77332:8
@ - 8
.4 @ + 37
PAIR (1.3 @ -1, 7765:8)
```

III. ARITHMETIC AND ARITHMETIC EXPRESSIONS

III.A ARITHMETIC EXPRESSION

Syntax:

```
<arithmetic expression> ::= <term>|<adding operator> <term>|
                            <arithmetic expression> <adding operator> <term>
<term> ::= <factor>|<term> <multiplying operator> <factor>
<adding operator> ::= + | -
<multiplying operator> ::= × | /
<factor> ::= <arithmetic primary>|<arithmetic primary>
             <exponentiation operator> <arithmetic primary>
<arithmetic primary> ::= (<arithmetic expression>)|<integer>|
                         <symbol>|<allocation counter designator>|
                         ABS (<arithmetic expression>)|
                         RELOC (<arithmetic expression>)|
                         SLA (<arithmetic expression>)|
                         WDA (<arithmetic expression>)|
                         RWA (<arithmetic expression>)
<allocation counter designator> ::= <space> @@ <space>
<exponentiation operator> ::= *
<space> ::= {one or more consecutive blank characters}
```

Semantics:

An arithmetic expression denotes a sequence of arithmetic operations to be performed (at assembly time) on certain specified quantities. The operations allowed are: addition, subtraction, multiplication, integer division and exponentiation (raised to the power of). Evaluation is performed in 24-bit two's complement.

ABS specified that the result of the evaluation of the arithmetic expression is to be made absolute (no matter what the relocatability of the expression turns out to be).

RELOC acts the same as ABS only the value is made relocatable.

SLA indicates that the parenthesized expression is to be evaluated using syllable arithmetic.

WDA indicates that the parenthesized expression is to be evaluated using word arithmetic.

RWA indicates that the parenthesized expression is to be evaluated using row arithmetic.

Examples:

1

(3)

X + 3

PLACEINMEMORY + Y/(2*(X-1)) + 2 × N

III.B ARITHMETIC WITH RELOCATABLE QUANTITIES

During the assembly of any particular code segment, it may not be known where in PE memory the object code will actually be loaded. Therefore, ASK must make provisions as it emits "object" code, for the placement of that code at an "arbitrary" place in PE memory. An "object" code file with that property is known as a relocatable code file. The assembly proceeds as if the code were to be loaded at PE memory location zero. At load time, however, the code may be loaded at PE memory address R. Therefore, if a PE symbol symbolizes location m at assembly time, it must symbolize location R+m at load time. Relocatable arithmetic takes the term R into account during the evaluation of arithmetic expressions.

In the following analyses:

Let R_s and R_s^1 stand for PE symbols which symbolize some PE memory address which may be relocated.

Let A_s and A_s^1 stand for either an integer or a symbol which symbolizes a PE memory address which may not be relocated. Henceforth a quantity of one of these two types shall be referred to as an absolute quantity.

Let m and m stand for the numbers associated with the symbols, and R stand for the starting PE memory address of the code at load time.

III.B.1 ADDITION

Three cases:

(1) $$R_s + R_s^1 = (R+m) + (n)$$

$$= R + (m+n)$$

This result is valid only for intermediate results. An expression which evaluates to a relocation amount greater than R is invalid and is flagged as such at assembly time.

(2) $$R_s + A_s = (R+m) + (n)$$

$$= R + (m+n)$$

This result is valid under all circumstances which allow a relocatable expression. The assembly time result is (m+n) as a relocatable quantity.

(3) $$A_s + A_s^1 = m + n$$

This result is the number (m+n) which is absolute (not relocatable) and as such is valid under any circumstances which allow absolute quantities.

III.B.2 SUBTRACTION

Four cases:

(1) $$R_s - R_s = (R+m) - (R+n)$$
$$= (R+R) + (m-n)$$
$$= m - n$$

The result of subtracting two relocatable quantities is an absolute quantity.

(2) $$R_s - A_s = (R+m) - n$$
$$= R + (m-n)$$

The result of subtracting an absolute quantity from a relocatable one is a relocatable quantity (m-n).

(3) $$A_s - R_s = n - (R+m)$$
$$= (n-m) - R$$

This result produces a negative relocation amount which, except as an intermediate result, is invalid.

(4) $$A_s - A_s = m - n$$

The result of subtracting one absolute quantity from another one is their difference (m-n), which is also absolute.

III.B.3 MULTIPLICATION

Three cases:

(1) $$R_s \times R_s = (R+m) \times (R+n)$$
$$= R^2 + R \times m + R \times n + m \times n$$

Multiplication of two relocatable quantities is invalid under all circumstances.

(2) $$R_s \times A_s = (R+m) \times n = (R \times n) + (m \times n)$$

Multiplication of a relocatable quantity and an absolute quantity is invalid under all circumstances.

(3) $A_s \times A_s = m \times n$

The only valid multiplication is that of two absolute quantities.

III.B.4 INTEGER DIVISION (All address arithmetic is integer arithmetic)

Four cases:

(1) $R_s / R_s = (R+m) / (R+n) = R / (R+n) + m / (R+n)$

Division of one relocatable quantity by another is invalid under all circumstances.

(2) $R_s / A_s = (R+m) / n = R/n + m/n$

Division of a relocatable quantity by an absolute quantity is invalid under all circumstances.

(3) $A_s / R_s = n / (R+m)$

Division of an absolute quantity by a relocatable quantity is invalid under all circumstances.

(4) $A_s / A_s = m/n$

The only valid division is that of two absolute quantities.

<u>Summary</u>:

The <u>valid</u> constructs in relocatable arithmetic are:

$R_s + R_s$	Valid only as an intermediate result.
$R_s + A_s$	Relocatable.
$A_s + A_s$	Absolute.
$R_s - R_s$	Absolute.
$R_s - A_s$	Relocatable.
$A_s - R_s$	Valid only as an intermediate result.
$A_s \times A_s$	Absolute.
A_x / A_s	Absolute.

An arithmetic expression is correct, with respect to relocatability, if the final result contains either the term $1 \times R$ (as in R_s) or $0 \times R$ (as in A_s). A further contextual restriction may be applied where only an absolute or only a relocatable result is valid.

Examples of Relocatable Arithmetic:

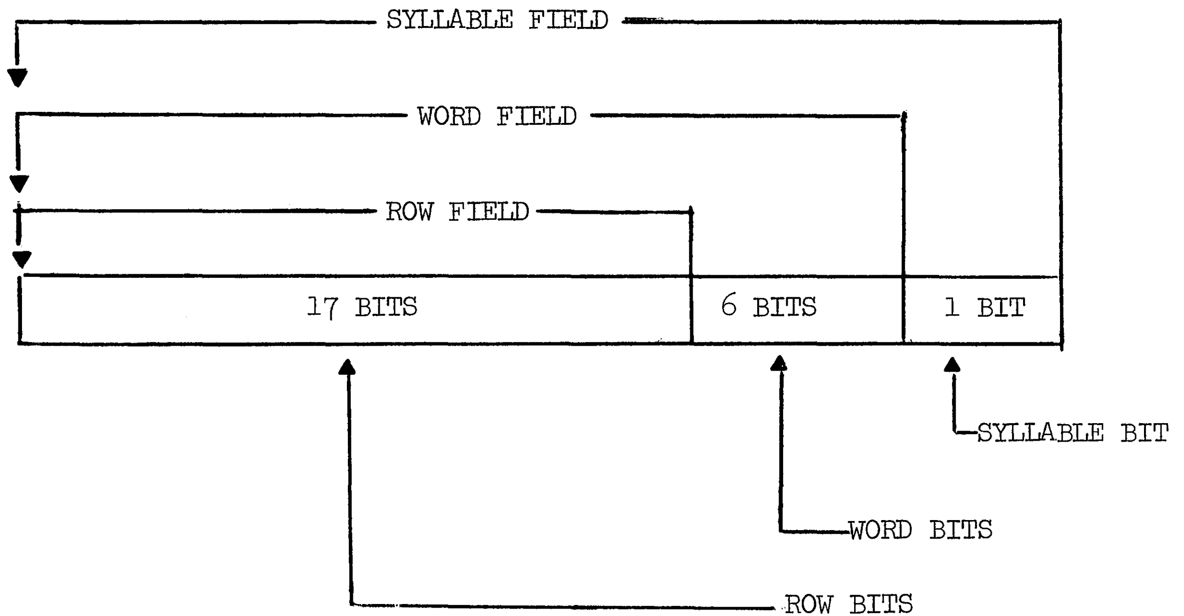
Let a symbol which begins with the letter "R" be understood to be relocatable, and one which begins with the letter "A" be understood to be absolute.

RX + RY + (-RA)	Relocatable.
AX - RY + RA	Absolute.
(RY - RY)/2	Absolute.
RX + RY	Invalid.
2 × RX	Invalid.
RX/2	Invalid.

III.C THREE MODES OF ARITHMETIC: SYLLABLE, WORD, AND ROW

ASK evaluates arithmetic expressions using one of three modes of arithmetic, depending upon context. Syllable arithmetic operates on a PE symbol as if it symbolizes the PE memory address of a 32-bit instruction syllable; word arithmetic operates on a PE symbol as if it symbolizes the PE memory address of a 64-bit word; row arithmetic operates on a PE symbol as if symbolizes the PE memory stack address of an entire row of 64-bit words across a quadrant. Since the same PE symbol may, at different times, appear in all three contexts, it would not be meaningful to use the same value for the symbol in each of the three modes of arithmetic.

For instance, a PE symbol PLACE which has the value 23 would represent three entirely different memory locations if the number 23 were used as a syllable address, word address, and row address. In order to avoid this ambiguity, ASK considers the value of a PE symbol to be divided into three fields for purposes of evaluating arithmetic expressions.



The above diagram represents the value of a PE symbol as it is interpreted by ASK. Syllable arithmetic operates on the syllable field; word arithmetic operates on the word field; row arithmetic operates on the row field.

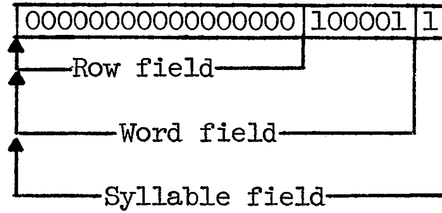
The interpretation of a numeric value depends upon how that value was specified in the source text:

1) The value of a PE symbol is interpreted as specified in the preceding paragraph.

2) The value of a CU symbol or a numeric constant is interpreted as designating the same field as the mode of arithmetic being performed on it. For example, the numeric constant 23 designates syllable, word and row 23 in syllable, word and row arithmetic, respectively.

Examples:

Suppose the PE symbol A has the value 67_{10}.



Syllable Arithmetic	A+2 = 69
Word Arithmetic	A+2 = 35
Row Arithmetic	A+2 = 2

IV. STRUCTURE OF AN ASK PROGRAM

Syntax:

<program> ::= BEGIN <compound statement>;
<end statement>.

<end statement> ::= <labeled end statement> |
<unlabeled end statement>

<labeled end statement> ::= <label list> <unlabeled end statement>

<unlabeled end statement> ::= END |
END <arithmetic expression>

<compound statement> ::= <statement> |
<compound statement> ; <statement>

Semantics:

The <end statement> indicates the end of the assembly language program. The appearance of the mnemonic END causes a halt instruction to be generated. A jump instruction is generated after the halt. If no arithmetic expression is present, the jump is to relocatable location 0. If there is an arithmetic expression present, the jump is to the location indicated by the value of the arithmetic expression (evaluated using word arithmetic) with the same relocatability as the value of the expression. The arithmetic expression on the relocatable location 0 as the case may be, should be the location of the first instruction to be executed.

V. STATEMENTS

Syntax:

```
< statement > ::= < unlabeled statement > |
                  < label list > < unlabeled statement > |
                  < ASK control statement >
< label list > ::= < label > : |
                   < label list > < label > :
< label > ::= < symbol >
< unlabeled statement > ::= < ILLIAC IV instruction > |
                            < ASK pseudo-op >

    < ILLIAC IV instruction > ::=

    AD          < PE address operand > |
    ADA         < PE address operand > |
    ADB         < PE address operand > |
    ADD         < PE address operand > |
    ADEX        < PE address operand > |
    ADM         < PE address operand > |
    ADMA        < PE address operand > |
    ADN         < PE address operand > |
    ADNA        < PE address operand > |
    ADR         < PE address operand > |
    ADRA        < PE address operand > |
    ADRM        < PE address operand > |
    ADRMA       < PE address operand > |
    ADRN        < PE address operand > |
    ADRNA       < PE address operand > |
    ALIT        < ACAR selector >   < short literal operand > |
    AND         < PE address operand > |
    ANDN        < PE address operand > |
    ASB         < blank PE operand > |
    BIN         < ACAR selector >   < CU memory operand > |
    BINX        < ACAR selector >   < CU memory operand > |
    CAB         < literal PE operand > |
    CACRB       < CU memory operand > |
    CADD        < ACAR selector >   < CU memory operand > |
```

CAND < ACAR selector > < CU memory operand > |

CCB < ACAR selector > < CU memory operand > |

CEXOR < ACAR selector > < CU memory operand > |

CHSA < blank PE operand > |

CLC < ACAR selector > < blank CU operand > |

CLRA < blank PE operand > |

COMPA < blank PE operand > |

COMPC < ACAR selector > < blank CU operand > |

COPY < ACAR selector > < CU memory operand > |

COR < ACAR selector > < CU memory operand > |

CRB < ACAR selector > < CU memory operand > |

CROTL < ACAR selector > < CU memory operand > |

CROTR < ACAR selector > < CU memory operand > |

CSB < ACAR selector > < CU memory operand > |

CSHL < ACAR selector > < CU memory operand > |

CSHR < ACAR selector > < CU memory operand > |

CSUB < ACAR selector > < CU memory operand > |

CTSBF < ACAR selector > < compare and skip operand > |

CTSBT < ACAR selector > < compare and skip operand > |

DUPI < ACAR selector > < CU memory operand > |

DUPO < ACAR selector > < CU memory operand > |

DV < PE address operand > |

DVA < PE address operand > |

DVM < PE address operand > |

DVMA < PE address operand > |

DVN < PE address operand > |

DVNA < PE address operand > |

DVR < PE address operand > |

DVRA < PE address operand > |

DVRM < PE address operand > |

DVRMA < PE address operand > |

DVRN < PE address operand > |

DVRNA < PE address operand > |

EAD < PE address operand > |

EOR < PE address operand > |

EQLXF < ACAR selector > < compare and skip operand > |

EQLXFA < ACAR selector > < compare and skip operand > |

EQLXT < ACAR selector > < compare and skip operand > |

EQLXTA < ACAR selector > < compare and skip operand > |

EQV < PE address operand > |

ESB < PE address operand > |

EXCHL < ACAR selector > < CU memory operand > |

EXEC < ACAR selector > < blank CU operand > |

FINQ < blank CU operand > |

GB < PE address operand > |

GRTRF < ACAR selector > < compare and skip operand > |

GRTRFA < ACAR selector > < compare and skip operand > |

GRTRT < ACAR selector > < compare and skip operand > |

GRTRTA < ACAR selector > < compare and skip operand > |

HALT < blank CU operand > |

IAG < PE address operand > |

IAL < PE address operand > |

IB < literal PE operand > |

ILE < PE address operand > |

ILG < PE address operand > |

ILL < PE address operand > |

ILO < blank PE operand > |

ILZ < blank PE operand > |

IME < PE address operand > |

IMG < PE address operand > |

IML < PE address operand > |

IMO < blank PE operand > |

IMZ < blank PE operand > |

INCRXC < ACAR selector > < blank CU operand > |

INR < blank CU operand > |

ISE < PE address operand > |

ISG < PE address operand > |

ISL < PE address operand > |

ISN < blank PE operand > |

IXE < PE address operand > |

IXG < PE address operand > |

IXGI < PE address operand > |

IXL < PE address operand > |

IXLD < PE address operand > |

JAG < PE address operand > |

JAL < PE address operand > |

JB < literal PE operand > |

JLE < PE address operand > |

JLG < PE address operand > |

JLL < PE address operand > |

JLO < blank PE operand > |

JLZ < blank PE operand > |

JME < PE address operand > |

JMG < PE address operand > |

JML < PE address operand > |

JMO < blank PE operand > |

JMZ < blank PE operand > |

JSE < PE address operand > |

JSG < PE address operand > |

JSL < PE address operand > |

JSN < blank PE operand > |

JUMP < short literal operand > |

JXE < PE address operand > |

JXG < PE address operand > |

JXGI < PE address operand > |

JXL < PE address operand > |

JXLD < PE address operand > |

LB < PE address operand > |

LDA < PE address operand > |

LDB < PE address operand > |

LDC < ACAR selector > < PE register specifier > |

LDD < register designator > |

LDE < mode pattern operand > |

LDE1 < mode pattern operand > |

LDEE1 < mode pattern operand > |

LDG < PE address operand > |

LDH < PE address operand > |

LD1 < PE address operand > |

LDJ < PE address operand > |

LDL < ACAR selector > < CU memory operand > |

LDR < PE address operand > |

LDS < PE address operand > |

LDX < PE address operand > |

LEADO < ACAR selector > < blank CU operand > |

LEADZ < ACAR selector > < blank CU operand > |

LESSF < ACAR selector > < compare and skip operand > |

LESSFA < ACAR selector > < compare and skip operand > |

LESST < ACAR selector > < compare and skip operand > |

LESSTA < ACAR selector > < compare and skip operand > |

LEX < PE address operand > |

LIT < ACAR selector > < long literal operand > |

LIT < ACAR selector > = < long literal operand > |

LOAD < ACAR selector > < CU memory operand > |

LOADX < ACAR selector > < CU memory operand > |

ML < PE address operand > |

MLA < PE address operand > |

MLM < PE address operand > |

```
MLMA      < PE address operand > |
MLN       < PE address operand > |
MLNA      < PE address operand > |
MLR       < PE address operand > |
MLRA      < PE address operand > |
MLRM      < PE address operand > |
MLRMA     < PE address operand > |
MLRN      < PE address operand > |
MLRNA     < PE address operand > |
MULT      < PE address operand > |
NAND      < PE address operand > |
NANDN     < PE address operand > |
NEB       < PE address operand > |
NOR       < PE address operand > |
NORM      < blank PE operand > |
NORN      < PE address operand > |
OFB       < blank PE operand > |
ONESF     < ACAR selector >   < skip operand > |
ONESFA    < ACAR selector >   < skip operand > |
ONEST     < ACAR selector >   < skip operand > |
ONESTA    < ACAR selector >   < skip operand > |
ONEXF     < ACAR selector >   < skip operand > |
ONEXFA    < ACAR selector >   < skip operand > |
ONEXT     < ACAR selector >   < skip operand > |
ONEXTA    < ACAR selector >   < skip operand > |
OR        < PE address operand > |
ORAC      < ACAR selector > |    < blank CU operand > |
ORN       < PE address operand > |
RAB       < literal PE operand > |
RTAL      < literal PE operand > |
RTAR      < literal PE operand > |
```

RTG < routing operand > |
RTL < routing operand > |
SAB < literal PE operand > |
SAN < blank PE operand > |
SAP < blank PE operand > |
SB < PE address operand > |
SBA < PE address operand > |
SBB < PE address operand > |
SBEX < PE address operand > |
SBM < PE address operand > |
SBMA < PE address operand > |
SBN < PE address operand > |
SBNA < PE address operand > |
SBR < PE address operand > |
SBRA < PE address operand > |
SBRM < PE address operand > |
SBRMA < PE address operand > |
SBRN < PE address operand > |
SBRNA < PE address operand > |
SETC < ACAR selector > < mode bit specifier > |
SETE < mode setting operand > |
SETE1 < mode setting operand > |
SETF < mode setting operand > |
SETF1 < mode setting operand > |
SETG < mode setting operand > |
SETH < mode setting operand > |
SETI < mode setting operand > |
SETJ < mode setting operand > |
SHABL < literal PE operand > |
SHABML < literal PE operand > |
SHABMR < literal PE operand > |

SHABR < literal PE operand > |

SHAL < literal PE operand > |

SHAML < literal PE operand > |

SHAMR < literal PE operand > |

SHAR < literal PE operand > |

SKIP < skip operand > |

SKIPF < skip operand > |

SKIPFA < skip operand > |

SKIPT < skip operand > |

SKIPTA < skip operand > |

SLIT < ACAR selector > < short literal operand > |

STA < literal PE operand > |

STB < literal PE operand > |

STL < ACAR selector > < CU memory operand > |

STORE < ACAR selector > < CU memory operand > |

STOREX < ACAR selector > < CU memory operand > |

STR < literal PE operand > |

STS < literal PE operand > |

STX < literal PE operand > |

SUB < PE address operand > |

SWAP < blank PE operand > |

SWAPA < blank PE operand > |

SWAPX < blank PE operand > |

TCCW < ACAR selector > < blank CU operand > |

TCW < ACAR selector > < blank CU operand > |

TXEF < ACAR selector > < compare and skip operand > |

TXEFA < ACAR selector > < compare and skip operand > |

TXEFAM < ACAR selector > < skip operand > |

TXEFM < ACAR selector > < skip operand > |

TXET < ACAR selector > < compare and skip operand > |

TXETA < ACAR selector > < compare and skip operand > |

```
TXETAM      < ACAR selector >   < skip operand > |
TXETM       < ACAR selector >   < skip operand > |
TXGF        < ACAR selector >   < compare and skip operand > |
TXGFA       < ACAR selector >   < compare and skip operand > |
TXGFAM      < ACAR selector >   < skip operand > |
TXGFM       < ACAR selector >   < skip operand > |
TXGT        < ACAR selector >   < compare and skip operand > |
TXGRA       < ACAR selector >   < compare and skip operand > |
TXGTAM      < ACAR selector >   < skip operand > |
TXGTM       < ACAR selector >   < skip operand > |
TXLF        < ACAR selector >   < compare and skip operand > |
TXLFA       < ACAR selector >   < compare and skip operand > |
TXLFAM      < ACAR selector >   < skip operand > |
TXLFM       < ACAR selector >   < skip operand > |
TXLT        < ACAR selector >   < compare and skip operand > |
TXLTA       < ACAR selector >   < compare and skip operand > |
TXLTAM      < ACAR selector >   < skip operand > |
TXLTM       < ACAR selector >   < skip operand > |
WAIT        < blank CU operand > |
XD          < PE address operand > |
XI          < PE address operand > |
ZERF        < ACAR selector >   < skip operand > |
ZERFA       < ACAR selector >   < skip operand > |
ZERT        < ACAR selector >   < skip operand > |
ZERTA       < ACAR selector >   < skip operand > |
ZERXF       < ACAR selector >   < skip operand > |
ZERXFA      < ACAR selector >   < skip operand > |
ZERXT       < ACAR selector >   < skip operand > |
ZERXTA      < ACAR selector >   < skip operand > |
```

Semantics:

A <label list> may be as many as 64 labels in length. Each label is given the same value. An <ILLIAC IV instruction> may be labeled with a <CU symbol>, but the user is advised that the value is truncated to 8 bits of precision. The value given to a <label> which labels an <ILLIAC IV instruction> is the value of the allocation counter (syllable bit included) at the time the <label list> is encountered, i.e., the location of the instruction being labeled.

The value given to a <label> which labels an <ASK pseudo-op> is defined individually for each pseudo. The user is advised to consult the section on pseudo operations for these definitions.

The <ASK control statement> is restricted in that it must begin in column one of a card or card image. Thereafter no restrictions are placed on the card format of a control statement except those which apply to card formats in general.

Detailed descriptions of each of the instruction mnemonics may be found in the ILLIAC IV Systems Characteristics and Programming Manual.

Restrictions:

The LDD instruction may address only RGB ($B).

The LDB instruction is the only instruction which may have RGD ($D) as its operand.

VI. REGISTER DESIGNATORS AND OPERAND FIELDS FOR CU INSTRUCTIONS

VI.A CU OPERAND FIELDS

Syntax:

```
<compare and skip operand> ::= <CU memory address specifier> <ACARX>
                               <skip field> <global-local specifier>
<CU memory operand> ::= <CU memory address specifier> <ACARX>
                        <global-local specifier>
<skip operand> ::= <skip field> <global-local specifier>
<blank CU operand> ::= <global-local specifier>
<short literal operand> ::= <arithmetic expression>|
                            = <arithmetic expression>
<long literal operand> ::= <number>|
                           <symbol>|
                           <index specifier>
<PE register specifier> ::= <PE register designator>
<mode bit specifier> ::= <mode bit>
<mode bit> ::= E1|E1|F1|F1|G|H|I|J
<ACAR selector> ::= <arithmetic expression>
<CU memory address specifier> ::= <arithmetic expression>|
                                  <CU register designator>
<skip field> ::= ,<arithmetic expression>
<global-local specifier> ::= ,G|,L|<empty>
<ACARX> ::= <arithmetic expression>|
            <empty>
<index specifier> ::= <arithmetic expression>,
                      <arithmetic expression>,
                      <arithmetic expression>
```

Semantics:

Operand fields for CU instructions provide a symbolic method of determining the value of each field of the instruction syllable except the op-code fields.

A <blank CU operand> sets no fields except the global/local field.

A <short literal operand> sets the low order 24 bits of the instruction to the value of the arithmetic expression.

A <long literal operand> sets the next 64 bits (two instruction syllables) after the LIT instruction to the value of the number, symbol or index specifier.

A <PE register specifier> encodes a PE register in the address field of the instruction.

A <mode bit specifier> encodes a mode bit in the address of the instruction.

The <ACAR selector> sets the ACAR field of the instruction to the value of the arithmetic expression.

A <CU memory address specifier> sets the address field to the value of the arithmetic expression or to the CU memory address of the indicated register.

The <skip field> sets the skip field of the instruction to a value which is determined as follows:

The expression is evaluated using syllable arithmetic. If the result is relocatable, ASK sets the skip field to a displacement such that the destination of the skip is the instruction whose address is the value of the expression. That is, if the expression were simply L and L were relocatable, a skip to L would be generated by ASK. If the result is absolute, ASK uses that value as the skip distance itself.

The <global-local specifier> indicates that the instruction being generated is to be flagged as global (G), local (L), or in the same global-local mode as the "rest" of the program (see explanation of pseudos GLOBAL and LOCAL).

The <ACARX>, if nonempty, sets the ACARX enable bit and bits 1:2 of the ACARX field to the value of the arithmetic expression modulo 4; otherwise, the ACARX field 0:3 is set to zero.

The <index specifier> indicates that 64 bits are to be set as three fields: bits 1:15, bits 16:24, and bits 40:24. These fields are set by the three arithmetic expressions respectively. Bit 0 of the 64 bits is not able to be set by this construct. In field one (bits 1:15), ASK

forms a 15-bit sign-magnitude representation of the arithmetic expression. In fields two and three the 24-bit two's complement value is inserted as is.

With the exception of the <skip field>, all arithmetic expressions are evaluated using word arithmetic. With the exception of the <skip field>, <short literal operand>, and fields two and three of the <index specifier>, arithmetic expressions must have an absolute result. The above-mentioned exceptions may have either a relocatable or an absolute value.

<u>Examples</u>:

Compare and skip operand:

```
.DELTA, IØØP
$C3 , L+1
.DELTA -1 (3) , LABEL, G
```

CU memory operand:

```
$D34 2 , L
N x .STUFF   PRESENTACARX
$TRI , L
```

Skip operand:

```
,LØØP
,L + 2
,- 14
,DESTINATION - ( @@ + 1 ),G
```

Blank operand:

```
,G
,L
```

Short literal operand:

```
= SUBRØUTINE
= 77777777:8
 2*(N-1)
```

Long literal operand:

```
= INCREMENT, LIMIT, INITIALVAL
= -1, 0, 64
= SCALEFACTØR
= 1.7325 @ 18
= 10000000000000000000000:8
= -(2*14-1), -1, -1
```

ACAR selector:

```
(REGISTER -1)
3
(2)
```

CU memory address specifier:

```
.LØCAL + 3 × (.Q-2*(N-1)+1)
$D2
$3D40
$C1
$ICR
$ACR
```

ACARX:

```
(REGISTER -1)
3
(2)
```

VI.B REGISTER DESIGNATORS IN CU

Syntax:

<CU register designator> ::= $<quadrant specifier> <register mnemonic>|
$<register mnemonic>

<quadrant specifier> ::= 0|1|2|3

<register mnemonic> ::= D0|D1|D2|D3|D4|D5|D6|D7|D8|D9|D10|D11|D12|D13|D14|
D15|D16|D17|D18|D19|D20|D21|D22|D23|D24|D25|D26|
D27|D28|D29|D30|D31|D32|D33|D34|D35|D36|D37|D38|
D39|D40|D41|D42|D43|D44|D45|D46|D47|D48|D49|D50|
D51|D52|D53|D54|D55|D56|D57|D58|D59|D60|D61|D62|
D63|C0|C1|C2|C3|ICR|ACR|ALR|AMR|AIN|MC0|MC1|MC2|
TRI|TRO

Semantics:

A <CU register designator> denotes an addressable register in the CU. Each CU register designator symbolizes the 8-bit encoding of the address of a register in CU memory. If the quadrant specifier is present, the leading two bits of the 8-bit field are assigned the specified number.

D0, D1, ..., D63 — Denote the 64 ADB locations.

C0, C1, C2, C3 — Denote the 4 ACAR registers.

The remaining register mnemonics denote the register which they abbreviate.

Examples:

$C0
$D32
$2D32
$ICR

VII. REGISTER DESIGNATORS AND OPERAND FIELDS FOR PE INSTRUCTIONS

VII.A PE OPERAND FIELDS

Syntax:

<blank PE operand> ::= <empty>

<PE address operand> ::= <ADR use indicator>
<address field> <ACARX> |
<address field> <ACARX> <ADR use> |
<register designator> <ACARX>

<literal PE operand> ::= <ADR use indicator>
<address field> <ACARX> |
<address field> <ACARX> <ADR use>

<routing operand> ::= <routing specifications> <ACARX>

<address field> ::= <arithmetic expression>

<ADR use indicator> ::= *|#|=|#*|*#

<ADR use> ::= ,<arithmetic expression> |
<empty>

<routing specifications> ::= <arithmetic expression> |
<arithmetic expression> <routing distance> |
<PE register designator> |
<PE register designator> <routing distance>

<routing distance> ::= ,<arithmetic expression>

Semantics:

The <address operand> specifies the ACARX, ADR use and address field for those PE instructions which specify an operand address.

The <literal operand> specifies the ACARX, ADR use and address field for those PE instructions which do not require an operand but, rather, a shift count or bit number encoded in the address field of the instruction.

The <routing operand> is used in conjunction with only two instructions, RTG and RTL.

The <address field> sets the 16-bit address field of the instruction to the value of the arithmetic expression. The expression is evaluated using row arithmetic and may be either relocatable or absolute.

The <ADR use indicator> sets the ADR use field of the instruction. The convention used is as follows:

Symbol	Bits	ADR USE FIELD 13	14	15	Meaning
*		0	1	1	RGX indexing
#		1	0	1	RGS indexing
#|#		1	1	1	Combined indexing
=		0	0	0	Literal

The <ADR use> sets the ADR use field of the instruction to the value of the arithmetic expression. Word arithmetic is used in evaluating the expression and the expression must be absolute. If the <ADR use> is <empty>, the ADR use field of the instruction is set to 1 (memory fetch--no indexing). Thus the ADR use field of the instruction may be set by either the <ADR use indicator> or <ADR use>.

A <register designator> causes one of two things to happen. If the specified register is a PE register, the ADR use field is set to 4 (register code) and the address field is encoded so as to specify the indicated register. If the specified register is an ACAR, the ADR use field is set to 0 (literal), the address field is set to 0 and the ACARX field is set to the indicated ACAR and the enable bit set.

The <routing specifications> indicates the register connectivity and routing distance for the route instructions. a) If a single <arithmetic expression> is used, ASK assembles a route of that distance, setting the register connectivity to the R register.

b) If the construct <arithmetic expression> <routing distance> is used, the first expression sets the register connectivity portion of the address field and the second sets the routing distance portion of the address field.

c) If only a <PE register designator> is used ASK sets the register connectivity portion of the address field to the indicated register and sets the routing distance portion of the address field to zero.

d) The construct <PE register designator> <routing distance> is self explanatory.

The <ACARX> sets the ACARX field enable bit of the instruction to one and encodes the ACAR indicated by the value of the expression (taken modulo 4). Word arithmetic is used to evaluate the expression and the expression must be absolute.

Examples:

Address operand:

```
* X-1 (2)
# P2 ACAR
* MATRIX + (Q - R)
STUFF  (2),3
MEMØRY,1
MEMØRY
= X + 14:8
= 0 (3)
$C3
$B
$R  (2)
```

Literal operand:

```
SHIFTCØUNT
BITNUMBER  (2),5
#BITNUMBER  (2)
*(SHIFTCØUNT)  (2)
```

Routing operand:

```
DISTANCE
2*WHICHREGISTER,DISTANCE
$S,DISTANCE
DIST  (2)
CHUZREG,DIST  (1)
$A,0  (2)
$A  (2)
```

Address field:

```
PQ
PDQ + 2*N
3
-1
```

ADR use:

```
,3
,WHICHØNE/2
,LITERAL + MAYBENØT
```

Routing specifications:

HERETØTHERE

REGISTER,24

$B,1

$A

Routing distance:

,DIST

,-1

,0

,NUMBERØFPES -1

VII.B REGISTER DESIGNATORS IN PE

Syntax:

<register designator> ::= $<register mnemonic>

<register mnemonic> ::= A|B|D|R|S|X|C0|C1|C2|C3

<PE register designator> ::= $<PE register mnemonic>

<PE register mnemonic> ::= A|B|R|S|D|X

Semantics:

A <PE register designator> denotes a register in the PE.

In addition, a <register designator> can denote the common data bus as defined by the contents of a specified ACAR. A <PE register designator> causes the encoding for that register to be placed in the address of the instruction. If a <register designator> specifies an ACAR, the address field of the instruction is set to zero, the ADR-USE field is set to zero (literal) and the ACARX field is set to the specified ACAR and the ACARX field enable bit is set.

Examples:

$A

$X

$C1

VIII. OPERAND FIELDS FOR MODE SETTING INSTRUCTIONS

Syntax:

<mode pattern operand> ::= <arithmetic expression> <ACARX> |
<ACAR designator>

<mode setting operand> ::= <left mode specifier> <mode operator>
<right mode specifier> <ACARX>

<ACAR designator> ::= $C0|$C1|$C2|$C3

<left mode specifier> ::= <mode bit> | -<mode bit>

<mode bit> ::= E|E1|F|F1|G|H|I|J

<mode operator> ::= AND|OR|.AND.|.OR.

<right mode specifier> ::= E|E1|-E|-E1

Semantics:

The <mode pattern operand> is used in conjunction with the mode-bit loading mnemonics (LD-). In these instructions, the ILLIAC IV hardware ignores the ADR use field, i.e., the address field is treated as a literal and is ACAR indexable.

The <mode setting operand> is used in conjunction with the mode setting mnemonics (SET-). The address field of the instruction is encoded for the same operation as is indicated by the operand field. The convention -<mode bit> means the logical negation of the specified mode bit.

If the mode operator AND or ØR are used a space must immediately precede and succeed them.

Examples:

Mode pattern operand:

```
1
0
0  (2)
$C2
```

Mode setting operand:

```
E ØR E1
I AND -E  (2)
H .ØR. -E1
```

IX. ASK PSEUDO OPERATIONS

IX.A GENERAL

Syntax:

```
<ASK pseudo-op> ::= EQU <EQU operand> |
                    SYL <SYL operand> |
                    WDS <WDS operand> |
                    BLK <BLK operand> |
                    FILL <FILL operand> |
                    SET <SET operand> |
                    DATA <DATA operand> |
                    ØRG <ØRG operand> |
                    CHWS <CHWS operand> |
                    REGP <REGP operand> |
                    REGC <REGC operand> |
                    SYN <SYN operand> |
                    GLØBAL |
                    LØCAL |
                    DEFINE <DEFINE part>
```

Semantics:

ASK pseudo operations are instructions directly to ASK which may or may not generate ILLIAC IV code. Each pseudo is discussed individually below:

IX.B EQU PSEUDO

Syntax:

```
<EQU operand> ::= <arithmetic expression> |
                = <long literal operand> |
                  <CU register designator>
```

Semantics:

The EQU pseudo operation must have a <label list>. The function of the pseudo operation is to assign a value to the symbol(s) in the label list. If the EQU operand is an arithmetic expression, the result of the expression (evaluated using word arithmetic) is put in the word field portion of the symbol's value. (refer to diagram p. 3-7). The syllable bit is set to zero. If a CU register designator is used, the symbol(s) receives the CU memory address of the indicated register in the word field portion of its value. If a long literal operand is used, a PE

symbol assumes the 64-bit value of the long literal operand; a CU symbol receives the rightmost 8 bits of the value of the operand, i.e., the value is truncated to 8 bits.

Restrictions:

All symbols in the label list must not have been previously defined.

All symbols referred to in the operand field must have been previously defined.

Examples:

```
ONE: EQU = 1.0
P:Q:R: EQU $D14
X:     EQU A - (B + 2 x N) where A,B,N have been previously
           defined
LOOPCONTROL: EQU = 1, 63, 0
```

IX.C SYL PSEUDO

Syntax:

<SYL operand> ::= <arithmetic expression> |
<empty>

Semantics:

The SYL pseudo operation serves to reserve a block of 32-bit syllables. A label list is optional. If any labels are present, they receive the value of the allocation counter at the time the SYL pseudo is encountered. ASK then emits the number of no-ops indicated by the value of the arithmetic expression (evaluated using word arithmetic), i.e., the requested block of 32-bit syllables is filled with no-ops. The value of the arithmetic expression must be absolute. If the <SYL operand> is <empty>, an expression value of zerø is assumed.

Examples:

```
X: SYL 31
CURRENTACVALUE: SYL
```

IX.D WDS PSEUDO

Syntax:

<WDS operand> ::= <arithmetic expression> |
<empty>

Semantics:

The WDS pseudo operation serves to reserve a block of 64-bit words, of length equal to the value of the arithmetic expression (evaluated using word arithmetic). The allocation counter is first adjusted to a 64-bit word boundary (even syllable), if necessary. If an adjustment is made, a no-op is placed in the syllable which is skipped over. At this point, all labels receive the value of the allocation counter (the label list is optional). The block of 64-bit words is then created by filling the appropriate number of words with zeroes. The allocation counter then points to the next available 32-bit syllable at the end of the block of 64-bit words. The value of the arithmetic expression must be absolute. If the <WDS operand> is <empty>, the expression value of zero is assumed.

Examples:

```
P: WDS
Q: WDS 64
   WDS
```

IX.E BLK PSEUDO

Syntax:

<BLK operand> ::= <arithmetic expression> |
<empty>

Semantics:

The BLK pseudo operation serves to reserve a block of 4096-bit "words", i.e., rows of 64-bit words across PE memory. The number of rows is determined by the value of the arithmetic expression (evaluated using word arithmetic). If necessary, ASK adjusts the allocation counter to a quadrant boundary, filling in no-ops if the adjustment has to take place. All labels then receive the value of the allocation counter. The requested number of "words" is then spaced over (inserting zeroes) and the allocation counter is set to the next available syllable beyond the requested block of storage. The allocation counter will point to a quadrant boundary after "execution" of this pseudo.

If the <BLK operand> is <empty>, the expression value of zero is assumed.

Examples:

```
X: BLK 64
   BLK
```

IX.F FILL PSEUDO

Syntax:

<FILL operand> ::= <arithmetic expression> |
<empty>

Semantics:

Let V be the value of the arithmetic expression. V determines a nonzero power of two, M, which is the smallest power of two not less than V. The directive to the assembler is to adjust the allocation counter to a position--syllable address--such that the allocation counter is congruent to V modulo M. Word arithmetic is used in evaluating the arithmetic expression. If the value of the expression is zero or if the operand field is empty, M is defined as being equal to 2. If the allocation counter has to move, no-ops are filled into the syllables skipped over. Labels are optional and, if any are present, receive as their value the value of the allocation counter after adjustment.

Examples:

	FILL 2	Even syllable
	FILL 7	Seventh syllable in a block of 8
X:	FILL 16	Head of a block of 16 syllables

IX.G SET PSEUDO

Syntax:

<SET operand> ::= <arithmetic expression> |
<empty>

Semantics:

See definition of EQU with an arithmetic expression as operand for the operation of SET. There are three differences:

1) No multidefinedness check is made on the symbols being defined, i.e., one or more symbols in the label field may have been previously defined,
2) The labels are redefined at the same points in the program in PASS II, and
3) If the operand field is empty, the symbol (s) is defined as the current value of the allocation counter.

The SET pseudo operation requires a label list. Word arithmetic is used in evaluating the arithmetic expression.

Examples:

```
        P:  SET 1
        P:  SET P+1
        HEREIAM:  SET
```

IX.H DATA PSEUDO

Syntax:

<data operand> ::= <data list>

<data list> ::= <data list element> |
<data list>, <data list element>

<data list element> ::= <number> | <symbol> | <string> |
(<data list>) <repeat part>

<repeat part> ::= <arithmetic expression>

Semantics:

The DATA pseudo operation provides for the loading of data into PE memory. A label list is optional. If necessary, the allocation counter is first adjusted to a word boundary and a no-op is inserted in the skipped syllable. The specified data is then placed in PE memory as 64-bit words.

If a number is used, its converted value (64-bit) is placed in memory.

If a symbol is used, the value of its syllable field is placed in memory, right justified, in a field of zeroes.

A repetitive list is placed in memory element by element, repeated as many times as is indicated by the value of the repeat part (word arithmetic).

Examples:

```
            DATA   -1
STUFF:      DATA    2,  3,  1.2,  01.3 @ -8,  (1, -1)  N-1,  X,  774:8
```

IX.I ORG PSEUDO

Syntax:

<ORG operand> ::= <arithmetic expression>

Semantics:

The ORG pseudo operation sets the allocation counter to the value of the arithmetic expression. Any labels are also given this value (in the syllable field). The expression is evaluated using syllable arithmetic. The allocation counter will have the same relocatability as the value of the expression, i.e., symbols defined by labeling an ILLIAC IV instruction will henceforth be absolute or relocatable, depending upon whether the value of this expression is absolute or relocatable.

Examples:

```
ORG @@ + 3
ORG X
```

IX.J CHWS PSEUDO

Syntax:

<CHWS operand> ::= <arithmetic expression>

Semantics:

The CHWS pseudo operation emits one ILLIAC IV instruction which sets the word size bit in the ACR register for 32 or 64 bit arithmetic in the PE's. The setting of this bit is according to the value of the arithmetic expression (word arithmetic).

Value of Expression	Word Size Setting Generated
0	64 bit
1	32 bit
32	32 bit
64	64 bit
Anything Else	Undefined

Examples:

```
CHWS  64
CHWS   1
```

IX.K REGP PSEUDO

Syntax:

<REGP operand> ::= <new PE register designator> = <register designator>

<new PE register designator> ::= $<identifier>

Semantics:

The REGP pseudo operation serves to rename register designators which may appear in the operand field of a PE instruction. This pseudo must not have a label list. At any point in the program after the appearance of the REGP pseudo-operation the defined <new PE register designator> may be used interchangeably with the <register designator> which defined it.

Restriction:

If any label of this pseudo operation is identical to a label used elsewhere in the program, the REGP pseudo which defines that label must be placed physically before the other definition of the label.

Examples:

```
REGP $AREGISTER = $A
REGP $BROADCASTNUM = $C1
```

Examples of uses of the defined register designators:

```
LDA    $BROADCASTNUM
RTL    $AREGISTER,3
```

IX.L REGC PSEUDO

Syntax:

<REGC operand> ::= <new CU register designator> <CU register designator>

<new CU register designator> ::= $<identifier>

Semantics:

The REGC pseudo operation serves to rename CU register designators which may appear in the operand field of a CU instruction. The operation of this pseudo is the same as that of the REGP pseudo.

Examples:

```
REGC  $INSTREG = $AIR
REGC  $COUNTER = $C1
```

Examples of Use:

```
LDL (0)  $INSTREG
STL (0)  $COUNTER
```

IX.M SYN PSEUDO

Syntax:

<SYN operand> ::= {any defined ILLIAC IV-op mnemonic or ASK pseudo-op mnemonics}

Semantics:

The SYN pseudo operation serves to make the label(s) of the label list (which must be present) synonymous with the ILLIAC IV instruction mnemonic or ASK pseudo-op mnemonic. At any point in the program after the appearance of the SYN pseudo operand, the defined label may be used interchangeably with the operation which defined it.

Examples:

```
MULTIPLY: SYN MLRN
DIVIDE:   SYN DVRN
DIV:      SYN DVM
```

IX.N GLOBAL PSEUDO

Semantics:

This pseudo-operation causes ASK to assemble CU instructions in the Global mode unless

1) A local pseudo-operation appears later, or
2) A CU instruction has a non-empty <Global-local specifier>, in which case that instruction only is assembled with the indicated Global-localness.

IX.O LOCAL PSEUDO

Semantics:

This pseudo-operation causes ASK to assemble CU instructions in the local mode unless

1) A global pseudo-operation appears later, or
2) A CU instruction has a non-empty <global-local specifier>, in which case that instruction only is assembled with the indicated global-localness.

IX.P DEFINE PSEUDO

Syntax:

<define pseudo> ::= DEFINE <define part>

<define part> ::= <define element> |
<define part>, <define element>

<define element> ::= <define identifier> =
<define text> ##

<define identifier> ::= <identifier>

<define text> ::= {any sequence of characters not including the character unless enclosed in string quotes}

Semantics:

The define pseudo causes the <define identifier> to serve as an abbreviation for the text bracketed by the = and the ##. From that point on in the program, whenever the <define identifier> is written, ASK will substitute for it the <define text> with which it is associated.

Restrictions:

1) The <define text> must not contain any unmatched " symbols.
2) A defined identifier may not appear as a PE or CU register mnemonic.
3) A defined identifier may be used alone as a <mode operand> but may not be used alone as a <left mode specifier> <mode operator> or <right mode specifier>.

Example:

Define

```
LASTWORD = FILL 126; WDS ##
       Y = 3 ##;
X: LASTWORD Y ;
```

is the same as:

```
X: FILL 126;  WDS 3 ;
```

X. ASK CONTROL STATEMENTS

Syntax:

<ASK control statement> ::= $<verb list>
<verb list> ::= <verb> | <verb list> <verb>
<verb> ::= <input specifier> |
<output specifier> |
<patch specifier> |
<option specifier>
<input specifier> ::= <input file designator> <label equation>
<input file designator> ::= CARD | TAPE1 | TAPE2 | TAPE3 | TAPE4 |
TAPE5 | TAPE6 | TAPE7 | TAPE8 |
TAPE9 | TAPE10 | TAPE11 | TAPE 12 |
TAPE13 | TAPE14 | TAPE15
<label equation> ::= <empty> |
= <multi-file id>/<file id> <disk or tape file>
<disk or tape file> ::= SERIAL | <empty>
<multi-file id> ::= <identifier>
<file identifier> ::= <identifier>
<output specifier> ::= <output file designator> <label equation>
<output file designator> ::= NEWDISK | NEWTAPE
<patch specifier> ::= MERGE <label equation> |
VOID <base ten number>
<option specifier> ::= LIST | SYNTAX |
XREF | BLOWUP | PUNCH |
SEQ | SEQ + <base ten number>

Semantics:

A <control statement> causes the assembler to change its mode of operation with respect to file handling or listing options.

An <input specifier> directs ASK to accept symbolic input from a file of the user's choice. The file CARD is the main input file for ASK, i.e., ASK must find its first input in file CARD. If ASK is directed to another input file, it assembles from that file until either it encounters a control card with an input specifier or reads the end of file marker. In the former case, ASK begins assembling from the new file and "remembers" which file it was assembling from. In the latter case, ASK closes the file from which the

EOF was read and continues assembling from the file which contained the control statement which directed it to the file it has just closed. Assembly proceeds from the card image immediately following the control statement in this case. If the <label equation> part is non-empty, ASK attaches itself to the specified tape or disk file.

If more than one <input specifier> are given in an <ASK control statement>, ASK will assemble from the file which is listed last until an EOF is reached. Then it will assemble from the file listed next to last until an EOF is reached. ASK will continue in the fashion until all input files listed on the control statement are exhausted. It will then go back to assembling the file in which the <ASK control statement> appeared.

An <output specifier> directs the assembler to create a new symbolic tape or disk file. This file will contain the totality of card images which ASK has processed from whatever files their origin may have been. Once an output specifier has been used, it is not necessary to specify it on subsequent control cards, since the option remains on for the rest of the assembly. It is possible, however, to direct ASK to create different output files for different sections of code by placing several control statements with an output specifier and label equation in the source file.

If the <patch specifier> is used, ASK considers the totality of card images from the files available as input file designators as an update deck for file MERGE. The functions of replacement, deletion and insertion are available. The selection criterion for which card image ASK will next process is the sequence number comparison between the next available card image from file MERGE and the designated input file. The selection algorithm is as follows:

	Relation between Sequence Numbers	File from Which Input is Taken
1)	"PATCH" sequence $<$ MERGE sequence	"patch"
2)	"PATCH" sequence $=$ MERGE sequence	"patch"
3)	"PATCH" sequence $>$ MERGE sequence	MERGE

In case 1), the card from the MERGE file is retained for subsequent comparisons. In case 2), the card from the MERGE file is discarded so that the next card from that file can be used for the next comparison.

In case 3), the card from the "patch" file is retained for subsequent comparisons.

If the VOID option is used, ASK discards card images from file MERGE as long as the sequence number from card images in file MERGE remains less than or equal to the value of the <base ten number>. The VOID is performed when its sequence number is less than or equal to the sequence number of the next card from file MERGE. Once ASK begins merging it continues to do so until the assembly is terminated or an EOF is read from file MERGE, at which point the user may choose to complete the assembly from the "patch" file or attach ASK to another file MERGE. The user may at any time attach ASK to another file MERGE through the use of the label equation construct.

At the time that a control statement is encountered, each of the options which may be an <option specifier>, except SEQ, is set to FALSE. The presence of the option specifier verb enables that particular option. The options and their effects are as follows:

LIST — The source program and instructions being generated are listed on the printer file.

SYNTAX — The generated object code is inhibited from being written into the object code file.

XREF — ASK is to cross-reference all identifiers, register designators, and control verbs as they are encountered and print out the cross-reference table at the end of the assembly.

BLØWUP — When printing the generated instructions, ASK will print all ILLIAC IV instructions in an "exploded view" with each field of the instruction displayed individually in octal separated from neighboring fields by a single space.

PUNCH — Causes ASK to punch each card image as it is processed. "Dollar cards", <ASK control statements>, are not punched.

SEQ — Causes ASK to resequence whatever source code output it is creating. The sequence increment is set equal to the value of the arithmetic term (evaluated using word arithmetic). If no term is given, a default value of 100 is used.

Examples:

Control statement:

```
$ LIST SYNTAX XREF
  NEWDISK = SØURCE/CØDE SERIAL
  SEQ + 1000

$ TAPE1 = SINE/RØUTINE SERIAL
  LIST X REF

$ LIST PUNCH SEQ + 10000

$ NEWTAPE

$ LIST VOID 19300 SYNTAX

$ TAPE1 = LAST/DONE TAPE5 = THIRD/DONE
          TAPE3 = SECOND/DONE  TAPE12 = FIRST/DONE
```

XI. PROGRAM DECK

Syntax:

< program deck > ::= < execution card > < file cards > < data card >
< program > < end card >

< execution card > ::= ?EXECUTE ASK/ASK II

< file cards > :: < empty > | < file cards > < file card >

< file card > :: ?FILE < file identifier > = < data file designator >
< medium >

< file identifiers > ::= < source identifier > |
< object code identifier > |
< symbolic output identifier >

< source identifier > ::= CARD | MERGE | TAPE1 ... | TAPE15

< object code identifier > ::= CODE

< symbolic output identifier > ::= NEWDISK | NEWTAPE

< data file designator > ::= < multi-file identifier > / < file identifier >

< medium > ::= SERIAL | < empty >

< data card > ::= ?DATA CARD | ?DATA < data file designator > | < empty >

< end card > ::= ?END

Semantics:

The < program deck > is the standard B5500 execution deck. More detailed information may be found in the Burroughs B5500 Electronic Information Processing System Operation Manual.

All illegal characters must appear in column one of the cards.

Note that the < file card > supplies information to the MCP whereas the < ASK control statement >, the "$ card", specifies file options to the assembler.

As stated in the discussion on < ASK control statement >, ASK must obtain its first input from the file card. By default the file CARD is the card reader. The < file card > may be used to direct the MCP to look for card on tape or disk, and if so, < data card > should also be < empty >. The < file card > could also be used to rename CARD, in which case the < data file designator > of the < file card > and of < data card > must be the same.

If many source files are to be used they may be "label equated" here or on an < ASK control statement >. In the former case, the information is given to the MCP, and in the latter to the assembler directly.

The file CODE is the object code file of <program> produced by ASK. It is a disk file as stated in the discussion on <ASK control statement>, the SYNTAX option will inhibit the object code from being put on this file. Naturally, the file CODE is not produced if a fatal error is encountered during the complication. If the file code is not label equated to a <data file designator>, it is written into the file ooooooo/CØDE which is a temporary file whose contents may be destroyed after the present job is completed.

<data card> should be <empty> only if CARD has been label equated to a tape or disk file.

Examples:

1) The following system control card sequence was used to generate a symbol disk file, SYMBOL/DISK, by assembling a disk file, DISK/FILE.

PROGRAM DECK	COMMENTS
? EXECUTE ASK/ASK II	
? FILE NEWDISK = SYMBOL/DISK	output disk file in source code
? FILE CARD = DISK/FILE SERIAL	input disk file
? END	

The first card image of DISK/FILE was

```
$ CARD LIST SYNTAX NEWDISK;
```

The SYNTAX option inhibited the object file CODE.

2) The following was used to assemble a program from the symbol tape, TAPE/INPUT, and generate the object code or a disk file MY/NAME.

PROGRAM DECK	COMMENTS
? EXECUTE ASK/ASKII	
? FILE CARD = TAPE/INPUT	input tape file
? FILE CODE = MY/NAME SERIAL	output disk file
? END	

Since there were no control statements in TAPE/INPUT the default $CARD LIST; was in effect.

3) The following sequence of system control card was used to generate object code onto the disk file, SAVE/DISK, from the card input file, READER.

PROGRAM DECK	COMMENTS
? EXECUTE ASK/ASK II	
? FILE CARD = READER	input card file
? FILE CODE = SAVE/DISK	output code file
? DATA READER	
$ CARD LIST XREF BLOWUP;	options specified
⋮ program ⋮	
? END	

4) In the following example a card file, a tape file, and a disk file are merged together. The output source code is placed on a type file, and the object code is placed on disk.

PROGRAM DECK	COMMENTS
? EXECUTE ASK/ASK II	
? FILE CARD = CARD/FILE	input card file
? FILE TAPE3 = TAPE/FILE	input tape file
? FILE TAPE10 = DISK/FILE SERIAL	input disk file
? FILE CODE = OBJECT/CODE	output object code
? FILE NEWTAPE = SOURCE/OUTPUT	output source code
? DATA CARD/FILE	
$ CARD LIST NEWTAPE;	
⋮ main ILLIAC IV program deck part 1 ⋮	card images are put on SOURCE/OUTPUT
$ TAPE10 LIST NEWTAPE;	TAPE10 file is opened and all of DISK/FILE is put onto SOURCE/OUTPUT
⋮ main ILLIAC IV program deck part 2 ⋮	card images are put on SOURCE/OUTPUT
$ TAPE3 LIST NEWTAPE	TAPE3 file is opened and <u>all</u> of file TAPE/FILE is put onto SOURCE/OUTPUT
⋮ main ILLIAC IV program deck part 3 ⋮	card images are put onto SOURCE/OUTPUT
? END	

At completion the output disk file CODE = OBJECT CODE has

[Main program part 1

[Contents of disk file, TAPE10

[Main program part 2

[Contents of tape file, TAPE3

[Main program part 3

XII. ERROR MESSAGES

ASK generates error messages when it encounters incorrect ILLIAC IV symbolic code. These messages are to be used by the programmer to help him in debugging his program. A list of the possible error message are given below. Their meaning is clear from the context. The only error which is not fatal is the omission of an end card.

ERROR MESSAGES

UNDEFINED OP-CODE

MULTIPLY DEFINED SYMBOL IN LABEL FIELD

UNDEFINED SYMBOL

IMPROPER LEFT MODE SPECIFIER

IMPROPER MODE OPERATOR

IMPROPER RIGHT MODE SPECIFIER

SKIP FIELD MISSING

SKIP DISTANCE TOO LARGE

-END CARD MISSING. INSERTED BY ASSEMBLER-

DISALLOWED CU MEMORY ADDRESS

THIS INSTRUCTION MAY NOT BE CARD INDEXED

CONTROL STATEMENT ERROR. NEXT INPUT FROM FILE CARD

TOO MANY LEFT PARENTHESES

TOO MANY RIGHT PARENTHESES

MULTIPLY DEFINED SYMBOL

RELOCATABLE ARITHMETIC WITH MULTIPLICATIVE OPERATOR

EXPRESSION YIELDS IMPROPER RELOCATION FACTOR

IMPROPER SEPARATOR

SEMICOLON MISSING OR TOO MANY FIELDS

THIS INSTRUCTION MAY NOT SPECIFY A REGISTER

IMPROPER PE REGISTER DESIGNATOR

NON-EMPTY OPERAND FIELD

IMPROPER CU REGISTER DESIGNATOR

NON-DIGIT APPEARS IN NUMBER

EXPONENT OVERFLOW

FILE IDENTIFIER TOO LONG

BASE SPECIFIER GREATER THAN 36

INTEGER TOO LARGE

THIS INSTRUCTION REQUIRES A LABEL

XIII. INDEX

< ACAR designator > 8-1
< ACAR selector > 6-1
< ACARX > 6-1
< adding operator > 3-1
< address field > 7-1
< ADR use > 7-1
< ADR use indicator > 7-1
< allocation counter designator > 3-1
< alphanumeric character > 2-1
< arithmetic expression > 3-1
< arithmetic primary > 3-1
< ASK control statement > 10-1
< ASK pseudo-op > 9-1
< base specifier > 2-2
< base ten digit > 2-2
< base ten number > 2-2
< blank CU operand > 6-1
< blank PE operand > 7-1
< BLK operand > 9-3
< character > 2-1
< CHWS operand > 9-6
< compare and skip operand > 6-1
< compound statement > 4-1
< CU memory address specifier > 6-1
< CU memory operand > 6-1
< CU register designator > 6-5
< CU symbol > 2-1
< data card > 11-1
< data file designator > 11-1
< data list > 9-5
< data list element > 9-5
< data operand > 9-5
< define element > 9-9
< define identifier > 9-9
< define part > 9-9
< define pseudo > 9-9
< define text > 9-9
< digit > 2-2
< disk or tape file > 10-1
< end card > 11-1
< end statement > 4-1
< EQU operand > 9-1
< execution card > 11-1
< exponentiation operator > 3-1
< exponent part > 2-2
< factor > 3-1
< file card > 11-1
< file cards > 11-1
< file identifier > 10-1
< file identifiers > 11-1
< FILL operand > 9-4
< global-local specifier > 6-1
< identifier > 1-3, 2-1
< ILLIAC IV instruction > 5-1
< index specifier > 6-1
< input file designator > 10-1
< input specifier > 10-1
< integer > 2-2
< integer part > 2-2
< label > 5-1
< labeled end statement > 4-1
< label equation > 10-1
< label list > 5-1
< left mode specifier > 8-1

< letter > 2-1

< literal PE operand > 7-1

< long literal operand > 6-1

< mantissa part > 2-2

< medium > 11-1

< mode bit > 6-1, 8-1

< mode bit specifier > 6-1

< mode operator > 8-1

< mode pattern operand > 8-1

< mode setting operand > 8-1

< multi-file id > 10-1

< multiplying operator > 3-1

< number > 2-2

< numeric character > 2-1

< object code identifier > 11-1

< option specifier > 10-1

< ORG operand > 9-6

< output file designator > 10-1

< output specifier > 10-1

< paired number > 2-2

< patch specifier > 10-1

< PE address operand > 7-1

< PE register designator > 7-4

< PE register mnemonic > 7-4

< PE register specifier > 6-1

< PE symbol > 2-1

< program > 4-1

< program deck > 11-1

< quadrant specifier > 6-5

< real number > 2-2

< real number or integer > 2-2

< REGC operand > 9-7

< register designator > 7-4

< register mnemonic > 6-5, 7-4

< REGP operand > 9-7

< repeat part > 9-5

< right mode specifier > 8-1

< routing distance > 7-1

< routing operand > 7-1

< routing specifications > 7-1

< SET operand > 9-4

< short literal operand > 6-1

< signed base ten number > 2-2

< signed real number > 2-2

< skip field > 6-1

< skip operand > 6-1

< source identifier > 11-1

< space > 3-1

< statement > 5-1

< SYL operand > 9-2

< symbol > 2-1

< symbolic output identifier > 11-1

< SYN operand > 9-8

< term > 3-1

< unlabeled end statement > 4-1

< unlabeled statement > 5-1

< unsigned real number > 2-2

< verb > 10-1

< verb list > 10-1

< WDS operand > 9-2