ComputerAutomation

NAKED MINI® Division

18651 Von Karman, Irvine, California 92715

Telephone: (714) 833-8830 TWX: 910-595-1767

OMEGA

ASSEMBLY SYSTEM

96007-00E3 August 1976

PRINTED IN THE U.S.A.

REVISION HISTORY

Rivision	Issue Date	Comments
A0	----	Original Issue
Al Thru E2		Misc. Updates
E3	August 1976	Eliminates references to Omega 3/05

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

Section 1

THE OMEGA ASSEMBLY SYSTEM

1.1 INTRODUCTION

This publication describes the assembler language for Computer Automation 16-bit minicomputers and millicomputers, and the three stand-alone programs which convert this language into object code.

OMEGA2 is the general-purpose assembler for all models of the LSI-2 and ALPHA-16. It runs on an LSI-2 (or an ALPHA-16) with a minimum configuration of 8K of memory and one ASR-33 Teletype (or an equivalent device).

Support is provided for this additional hardware:

Memory, to a maximum of 32K
Card Reader
Line Printer
High Speed Paper Tape Reader/Punch

OMEGA3 is a cross-assembler -- a variant of OMEGA2 which can be executed only on an LSI-2, but which generates object code executable only on an LSI-3/05. The paper tape Object Program is usually loaded into a 3/05 with the LAMBDA3 Object Loader.

Because the source language defined for these two programs is identical, this publication uses the name OMEGA or the phrase "the assembler" to denote whichever assembler is being used to accomplish the translation from Source Program to Object Program, and designates the assembler by name -- OMEGA2 or OMEGA3 -- only when there is, in fact, a meaningful distinction to be made.

OMEGA is called an "assembly system" because it includes a conversational Source Program editor, as well as a two-pass assembler. A Source Program can be constructed in memory, either from pieces of existing programs, or from scratch, and then assembled. The new Source Program, the corresponding Object Program, or both, can be punched out for future use.

Editing commands are entered thru the Teletype; listing and punching can be directed dynamically to any attached device. Input can be switched back and forth from the Teletype keyboard to a card reader, to a paper tape reader, and to the memory containing the newly-constructed Source Program.

The OMEGA editing commands are described in detail in Section 12, which includes a Command Summary chart suitable for use at the Teletype.

1.2 ASSEMBLER DEVICES

Source Input Device

During the editing or assembly of a program, OMEGA obtains statements to be processed from the Source Input Device.

The maximum length of a Source Input record is 80 bytes. A keyboard or paper tape record is terminated with a Carriage Return; extraneous Line Feeds, Rubouts, and Nulls are dropped during input.

Input supplied during the editing process is terminated with a record which starts with an Up-Arrow or a Slash. Input supplied during the assembly process is terminated with an END statement.

If a Source Input record starts with an Up-Arrow, OMEGA halts. The operator then readies another segment of input on the same device, and hits the RUN switch.

Listing Device

The assembly process, and the List command during the editing process, generate printed output on the Listing Device. A page of output is usually 66 lines of 72 characters each. These values can be changed, as explained under Omega Program Variables.

Sections 10 and 11 contain a sample Assembly Listing, and a detailed explanation of the layout.

Punch Output Device

The generation of an Object Program, and the Punch command during the editing process, generate records on the Punch Output Device.

A punched Object Program is in a format acceptable to LAMBDA2 or LAMBDA3. Section 7 explains why some tapes may not be directly usable by BLD or Autoload without first being processed thru LAMBDA.

A punched Source Program is in a format acceptable to OMEGA thru the Source Input Device.

1.3 SYNTAX NOTATION

This reference manual adopts a familar meta-linguistic notation to specify the valid syntax for each type of source statement. Each statement type is displayed as if it were a card located flush with the left edge of the narrative text; the distinction between the various fields will be self-evident from their contents and horizontal spacing.

Syntax elements which begin with a capital letter, but are otherwise in lower case, are generic terms, and are explained in the corresponding narrative.

A syntax element in upper case is a fixed part of the language.

An element surrounded by square brackets is optional.

A vertical stack indicates a choice of one entry from the stack.

Three periods following a right square bracket indicate an arbitrary repetition of the contents of the last pair of brackets.

The following syntax chart illustrates the complete notation:

```
[Label]     MNEM     [Operand[,Operand]...     [Comments]]
```

1.4 SOURCE STATEMENT FORMAT

Each source statement occupies the first 72 bytes of an isolated logical input record; any bytes remaining are discarded. Each statement is in the usual free-form arrangement --four variable-length fields delimited by blank columns.

Label Field

The Label Field starts in Column 1 of each source statement. If Column 1 is blank, then the Label Field is said to be empty, and ends with the first non-blank character -- that is, with the start of the Operation Field.

If Column 1 is not blank, then every column up to the next blank is either a Label or some type of assembler directive, such as a Comment Line, a New Page, or a New Op Code Definition.

If Column 1 is an alphabetic character, then the field contains a Label -- the name of a symbol or variable. The alphabetic character may be followed by 0 thru 5 alphanumeric characters, followed in turn by at least one more blank.

Operation Field

The Operation Field starts with the first non-blank column after the Label Field. It contains a character string identical in structure to a Label -- 1 to 6 alphanumeric characters, the first of which must be alphabetic. This string is called a Mnemonic, and indicates a machine instruction, a New Op Code, or an assembler directive.

Except for a directive, any Mnemonic can have its meaning changed at any point thru facilities built into the assembler language.

At least one blank column must follow the Mnemonic; an arbitrary number of blanks may be used to separate the Operation Field from the next field.

Operand Field

The existence of the Operand Field depends upon the definition of the Mnemonic used in the Operation Field. For some Mnemonics, no operands are meaningful, and the assembler never processes any source statement columns to the right of the Operation Field. For other Mnemonics, one or more operands are always required, and the assembler expects them to start with the first non-blank column after the Operation Field.

There are two types of statements which sometimes have an Operand Field, and sometimes do not:

END directives
LPOOL directives

For these, the programmer must either supply an Operand Field, or leave the rest of the source statement blank.

Each operand is of arbitrary length, and is determined by the nature of the source statement involved; the only restrictions are:

1. Single Quote characters must be paired.

2. Blanks and commas cannot occur outside of quoted text strings.

3. The last operand cannot extend past Column 72. The assembler does not allow continuation of the Operand Field onto another logical input record.

Each operand is separated from the next by a comma, and the last operand -- unless it extends to Column 72 -- must be followed by at least one blank column.

Comments Field

The Comments Field starts with the first non-blank column after the previous field, and extends to the rightmost column of the source statement. The assembler does not process the Comments Field, except to align it for a formatted listing.

If a given Mnemonic always requires an Operand Field, the Comments Field is not shown on syntax charts in this publication, because it cannot affect the validity of a statement.

If a Mnemonic never involves an Operand Field, the syntax chart may show the generic element Comments to emphasize that no operands are recognized.

For the few statement types which allow a Comments Field only if an Operand Field is also present, the syntax chart will show this construction:

```
[Label]     Mnemonic     [Operand     [Comments]]
```

Statement Fields as Listed

The assembler reformats each source statement before listing it, to provide uniform, more readable columns. If the source statements are keypunched on 80-column cards, the usual coding practice is to use the same fixed columns maintained on the listing:

01 -- 06	Label Field
07	Blank
08 -- 13	Operation Field
14	Blank
15 -- 72	Operand Field
24 -- 72	Comments Field (if Column 23 is blank)
73 -- 80	Discarded on Input

Section 2

OPERAND EXPRESSIONS

Each operand of an assembler language source statement may be a simple term -- a number or name -- or it may be a complex expression -- a formula consisting of several terms and operators.

This section is devoted to the various ways of coding terms and expressions. Subsequent sections will refer to the categories established here.

2.1 TERMS

A term may be characterized in several different ways:

Self-Defining or Symbolic
Defined or Undefined
Absolute or Relocatable

2.1.1 Self-Defining Terms

A self-defining term represents an immediately available value in one of these notations:

Decimal Number
Octal Number
Hexadecimal Number
Character Value

Decimal Numbers

A decimal number consists of 1 thru 5 decimal digits. It is distinguished from an octal number by having no leading zeros. The largest acceptable decimal number is 32767.

Octal Numbers

An octal number consists of 1 thru 7 octal digits -- the characters 0 thru 7. It is distinguished from a decimal number by having at least one leading zero. The largest acceptable octal number is 0177777.

Hexadecimal Numbers

A hexadecimal number consists of 1 thru 4 hexadecimal digits -- the characters 0 thru 9 and A thru F. It is distinguished from a symbolic term by having a colon prefixed. The largest acceptable hexadecimal number is :FFFF.

Character Values

A character value consists of 1 or 2 ASCII characters. The value is delimited with a preceding and a following Single Quote (or Apostrophe) character. If a Single Quote is actually part of the value, it must be represented by two successive Single Quotes. Printable ASCII characters, and their corresponding hexadecimal values, are charted in Appendix A of this publication.

Here are some examples of self-defining terms:

Decimal Numbers:

```
1
70
777
32000
```

Octal Numbers:

```
0
03
0777
```

Hexadecimal Numbers:

```
:0
:E
:64
:0FF
:FFFF
```

Character Values:

```
'A'
'*'
'XX'
'  '
'T '
' T'
```

2.1.2 Symbolic Terms

A symbol is the name of a value defined by the assembly process. Ordinarily, a symbol consists of 1 thru 6 alphanumeric characters. As in most programming languages, the first character of a symbolic name must be alphabetic -- that is, in the ASCII character range A thru Z.

The assembler accepts embedded colons in symbolic names, but the use of colons is reserved for CA-supplied software.

One symbolic name has a special construction. An isolated character $ -- or Currency Symbol -- represents the current value of the Location Counter at the point where the $ is referenced.

2.1.3 Defined Terms

A defined term has a value known to the assembler. A self-defining term is, of course, defined by its own representation. At any point within an assembly, a term is predefined if its nominal value has already been conclusively determined. The nominal value of a symbol is the value it will have after load processing if the relocation bias is specified to be zero.

Each use of a symbol before it becomes defined is called a forward reference. Because the assembler performs two passes over the Source Program, forward references are allowed in almost all contexts. However, certain directives which control Pass 1 processing will accept only predefined terms.

A symbol may be declared External by certain directives. An External symbol is considered a kind of forward reference which does not become defined until load time. An External reference may be used in certain restricted contexts, as specified in the detailed descriptions of each assembly language feature.

2.1.4 Undefined Terms

If a symbolic name is found to be neither defined, nor declared External, at the end of an assembly, it is considered undefined. Reference to an undefined term is usually an error, and the source statement is flagged on the listing.

Undefined terms may appear without error in SPAD statements, and in statements skipped by an IFT False or an IFF True.

2.1.5 Absolute Terms

An absolute term has the same value during the assembly as it will have after load processing, regardless of the relocation bias specified to the loader. It follows that self-defining terms are always absolute.

Symbolic terms are established as absolute if they are defined in certain ways. For example, a symbol defined thru a SET or EQU to an absolute expression is absolute. Similarly, a symbol defined as the Label of a statement within range of an ABS directive is absolute.

2.1.6 Relocatable Terms

A relocatable term has a nominal value during the assembly, but the value is subject to change during load processing. It follows that Externals are always considered relocatable.

Symbolic terms are established as relocatable if they are defined in certain ways. For example, a symbol defined thru a SET or EQU to a relocatable expression is relocatable. Similarly, a symbol defined as the Label of a statement within range of a REL directive is relocatable.

2.1.7 Unary Operators

The value represented by a term, whether self-defining or symbolic, may be adjusted by a unary operator prefixed to the term when it is the first in an expression.

Unary Plus (+)

A + character prefixed to a term has no effect upon its value. It may be used to emphasize that a term does not have a Unary Minus prefixed, or for any similar clarification of the source statement.

Unary Minus (-)

A - character prefixed to a term indicates 2's complementation of the signed arithmetic value of the term.

Here are some examples of unary operators:

Expression	Word Value in Hex
1	:0001
+1	:0001
-1	:FFFF

Assume that WN is a relocatable symbol with a nominal value of +1:

WN	:0001
+WN	:0001

This expression is an error, because it violates the rules explained under Absolute and Relocatable Expressions:

```
-WN
```

2.2 COMPLEX EXPRESSIONS

Terms are combined into complex expressions by using binary operators. An expression is always evaluated from left to right.

As expression evaluation proceeds from left to right, the current partial result of the evaluation, or intermediate value, is maintained as 16-bit binary number. An incoming term is limited to a 16-bit absolute or 15-bit relocatable value, as is the final evaluated result, or expression value.

As relocatable terms enter the expression evaluation, they cause the intermediate value to fluctuate between absolute and relocatable, according to rules explained in a following section. The nature of the final result determines whether the entire evaluated expression is called an absolute expression or a relocatable expression, and whether its Load Attribute is Absolute or Relocatable.

To clarify the discussion which follows, these symbols are adopted:

V	The intermediate value of the expression evaluation process
T	The leftmost unevaluated term, about to enter the expression evaluation
ABS	Any absolute value, either intermediate or final
REL	Any relocatable value, either intermediate or final

2.2.1 Binary Operators

Addition (V+T)

The expression V+T indicates the arithmetic addition of the values of V and T.

Subtraction (V-T)

The expression V-T indicates the arithmetic subtraction of the value of T from V.

2.3 ABSOLUTE AND RELOCATABLE EXPRESSIONS

As expression evaluation proceeds, an assembler artifact called R (for Relocation Factor) is associated with the current intermediate value V. At any point in the evaluation, R has some signed numeric value.

It is the manipulation of R which determines whether or not an expression is acceptable to the assembler, and whether the final expression is absolute or relocatable.

These are the rules for determining R at any intermediate or final point.

1. Set the initial value of R to 0.
2. If the very first term of the expression is relocatable, set R = 1. For -REL, set R = -1.
3. As the evaluation proceeds, for each V+REL, set R = R+1.
4. For each V-REL, set R = R-1.

At any point, R = 0 indicates that the intermediate or final value is absolute.

If R is not 0, the intermediate or final value is relocatable.

When the evaluation is completed, R must be either 0 or 1. Any other final R is an error.

2.4 OPERAND EXPRESSION PREFIXES

For some classes of machine instructions and assembler directives, the entire operand expression may be immediately preceded by certain characters which indicate a machine Addressing Mode. The effect of each prefix is held off until the assembler has obtained a final expression value.

The prefix characters are:

*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed

The assembler also accepts this special prefix for Word Reference operands only:

=	Literal Pool Reference

This prefix cannot be used for Byte Reference instructions. Refer to Sections 3 and 8 for details.

Section 3

CODING MACHINE INSTRUCTIONS

This section presents the valid assembler language syntax for each standard machine instruction. The instructions are divided into Syntax Classes, corresponding to the number of operands and to the Addressing Modes which are meaningful at machine level.

Syntax Class	Machine Function
1	Word Reference
2	Byte Immediate
3	Conditional Jump
4	Single Register Bit Change
5	Register and Control
6	Input/Output
7	Double Register Bit Change
8	Byte Reference
9	Double Register Arithmetic
10	Stack Reference

For each class, the rules for the source statement Operand Field are specified. Examples are given, to aid the programmer in visualizing the connection between an abstract syntax chart and a real Source Program.

An alphabetical list of every standard machine instruction mnemonic -- and which Syntax Class it falls into -- is included in this publication as Appendix B.

The machine instruction functions are described in the relevant Computer Handbooks:

LSI-2 Series Minicomputer Handbook, Publication 91-20400-00
LSI-3/05 Series Millicomputer Handbook, Publication 91-10005-00

3.1 CLASS 1: WORD REFERENCE

```
[Label]     Mnemonic  [ *  ]  Operand
                      [ @  ]
                      [ *@ ]
                      [ =  ]
```

Operand Field

Exactly one expression.
Any absolute or relocatable value.
External allowed.

Addressing Mode Prefix

No Prefix	Direct
*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed
=	Literal Pool Reference

Examples

1. Direct:

```
        LDA     :34
        STA     ABC+2
```

2. Indirect:

```
        LDA     *:34
        STA     *PTR
```

3. Indexed:

```
        LDA     @:34
        STA     @TABLE
```

4. Indirect Post-Indexed:

```
        LDA     *@:34
        STA     *@PTR
```

5. Literal Pool Reference:

```
        LDA     =1000
        LDX     =TABEND-TABLE
```

3.2 CLASS 2: BYTE IMMEDIATE

```
[Label]     Mnemonic    Operand
```

Operand Field

Exactly one expression.

Any absolute value equivalent to the range :00 thru :FF.

External not allowed.

Examples

1. Self-defining decimal operand:

```
            CAI         16
```

2. Self-defining character value operand:

```
            CAI         'Z'
```

3. Symbolic Operand:

```
BANG        EQU         '!'
            CAI         BANG
```

3.3 CLASS 3: CONDITIONAL JUMP

```
[Label]     Mnemonic    Operand
```

Operand Field

Exactly one expression.
(For special case of LSI-2 mnemonic JOC, refer to Appendix C)

Any absolute or relocatable value in the range

$-63 thru $+64

External not allowed.

Examples

1. Symbolic operand:

```
            JAZ         PARTY
```

2. Explicit relative location:

```
            JAZ         $-7
```

3.4 CLASS 4: SINGLE REGISTER BIT CHANGE

```
[Label]     Mnemonic     Operand
```

Operand Field

Exactly one expression.

Any absolute value, within the limits of the instruction function:

```
    0 thru 15 for BAO and BXO
    1 thru  6 for SIN
    1 thru  8 for Shifts
```

External not allowed.

Examples

1. Self-defining operand:

```
            LRA         6
```

2. Symbolic operand:

```
SZ          EQU         7
            LRA         SZ
```

3.5 CLASS 5: REGISTER AND CONTROL

[Label] Mnemonic [Comments]

Operand Field

None. Comments may immediately follow the Operation Field.

Examples

1. Label, mnemonic, no operands, comments:

```
COPY        TXA          TRANSFER X TO A
```

3.6 CLASS 6: INPUT/OUTPUT

```
[Label]    Mnemonic    Operand [,Operand]
```

Operand Field

Either 1 or 2 operands.

Each operand must be an absolute value.

Externals not allowed.

If only 1 operand is used, its value specifies the combined bits of the Device Address and Function Code.

If 2 operands are used, the first specifies the 5-bit Device Address, and the second specifies the 3-bit Function Code.

Examples

1. One hex operand:

```
            SEA         :3C
```

2. Two decimal operands:

```
            SEA         7,4
```

3. Two symbolic operands:

```
TTY         EQU         7
INIT        EQU         4
            SEA         TTY,INIT
```

3.7 CLASS 7: DOUBLE REGISTER BIT CHANGE

```
[Label]     Mnemonic     Operand
```

Operand Field

Exactly one expression.

Any absolute value, from 1 to 16.

External not allowed.

Examples

1. Self-Defining Operand:

```
            LRR          6
```

2. Symbolic Operand:

```
SZ          EQU          7
            LRR          SZ
```

3.8 CLASS 8: BYTE REFERENCE

```
[Label]     Mnemonic   [ *  ]
                       [ @  ] Operand
                       [ *@ ]
```

Operand Field

Exactly one expression.

Any absolute or relocatable value, except for the cases described on the next page.

External not allowed.

Addressing Mode Prefix

No Prefix	Direct
*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed

Expression Evaluation for Class 8

Each self-defining term represents a byte address value.

```
            LDAB        :04
```

addresses the 4th byte of memory.

Each symbolic term represents a word address value, and is multiplied by 2 before expression evaluation:

```
Q           EQU         7
FLD         TEXT        'WXYZ'
            LDAB        Q
            STAB        FLD
```

The LDAB addresses the 7th word of memory, or the 14th byte. Similarly, the word value of FLD, whether absolute or relocatable, must be doubled to produce a byte value.

```
            LDAB        FLD+3
```

addresses a location 3 bytes after the byte location of FLD -- the character 'Z' in the assembled text.

Operand Locations Not Acceptable

For reasons explained in the section on Scratchpad Literals, the assembler rejects a Byte Reference instruction which attempts Explicit Indirect Addressing of a location which is beyond Direct Addressing Range:

```
            xxxB        *ABSBIG
```

in which ABSBIG is Absolute, but higher than directly addressable Scratchpad;

```
            xxxB        *RELFAR
```

in which RELFAR is Relocatable, but beyond Direct Relative Addressing Range of the Byte Reference instruction.

Examples

1. Direct:

```
            LDAB        :34
            STAB        ABC+2
```

2. Indirect:

```
            STAB        *PTR
PTR         BAC         CHAR+1
```

3. Indexed:

```
            LDAB        @:34
            STAB        @TABLE
```

4. Indirect Post-Indexed:

```
            LDAB        *@:34
```

(At Word Location :34)

```
            BAC         CHAR+1
```

3.9 CLASS 9: DOUBLE REGISTER ARITHMETIC

```
[Label]     Mnemonic     [*]Operand
```

Operand Field

Exactly one expression.

Any absolute or relocatable value.

External allowed.

Addressing Mode Prefix

No prefix	Direct
*	Indirect Address

Examples

1. Direct:

```
            MPY         JKL+3
```

2. Indirect:

```
            DVD         *DVSR
```

3.10 CLASS 10: STACK REFERENCE

$[\text{Label}] \quad \text{Mnemonic} \quad \text{Operand}\begin{bmatrix} ,@ \\ ,+ \\ ,- \end{bmatrix}$

Operand Field

Exactly one expression, optionally followed by an Addressing Mode Specification.

Any absolute or relocatable value.

External allowed.

Addressing Mode Specification

No specification	Direct (Value of Pointer)
,@	Indexed (Pointer + Index Register)
,+	Pop (Increment Pointer After Access)
,-	Push (Decrement Pointer Before Access)

Examples

1. Direct:

```
          EMAS      STK
```

2. Indexed:

```
          IORS      STK,@
```

3. Pop:

```
          LDAS      STK,+
```

4. Push:

```
          STXS      STK,-
```

Section 4

ASSEMBLER CONTROL

The types of statements described in this section are not machine instructions, but directives -- they cause the assembler itself to take some action, or to recognize certain information presented to it.

The result is some variation in the assembly process -- either in the way the Source Program is translated, or in the appearance of the Assembly Listing, or both.

End of Source Program (END)

[Label] END [Operand [Comments]]

This directive terminates the assembly of the Source Program.

If a Source Program contains at least one LPOOL statement, a Literal Pool may be allocated by the assembler when an END is reached. The Pool will appear on the listing, and in the generated object code, before the END. Further details may be found in the section on Literal Pools.

The optional label of an END statement has the current value and Load Attribute of the Location Counter, after any Literal Pool generation. Unless a currently effective Location Control Directive has disturbed the continuity of the object code -- for example, a backward ORG -- the label on an END is the address of the first word following the end of the Object Program.

The optional operand specifies an execution-time Transfer Address. The operand may be any absolute or relocatable expression with predefined terms, except that reference to an External is not allowed.

The assembler communicates the Transfer Address -- or the fact that one was not specified -- to the loader. When a program is executed, the resolved Transfer Address receives initial control.

If several different Transfer Addresses are available in a number of Object Programs being loaded together, the loader will use the last Transfer Address processed.

The programmer should observe that no Comments may be used in an END statement which has no Operand.

End of Input Segment (up-arrow)

↑ Comments

This directive indicates the end of the current physical segment of the Source Input. The directive consists of an up-arrow in Column 1 of a source statement. The assembler displays "PAUSE" and halts the computer. The operator readies the next segment of Source Input and hits RUN.

Heading Title (TITL)

```
            TITL        Title
```

This directive supplies the title which appears in the page heading of the assembly listing. Starting exactly one blank after the last letter of TITL, the remaining characters of the source statement are taken to be the desired title. The title is initially blank, and each new title completely replaces the previous one.

A TITL statement is never listed. At the point where it would have appeared on the listing, the effect of a New Page directive is simulated.

New Page (period)

```
.Comments
```

This directive causes the next line listed to appear on a new page if at least 3 lines have appeared on the current page. It consists of a period in Column 1 of the source statement. The statement itself is never listed, and the Comments are ignored.

Comment Line (asterisk)

```
*Comments
```

A Comment Line appears on the assembly listing, but is not otherwise processed. The directive consists of an asterisk in Column 1 of the source statement. Any combination of printable characters and blanks may follow.

Machine Instruction Set (MACH)

MACH Operand

This directive is meaningful only for a Source Program assembled with OMEGA2, not with OMEGA3. It specifies the machine for which the program is intended, so the assembler can disallow those standard machine ~~ins~~truction mnemonics which would not be meaningful.

Each disallowed Mnemonic is flagged "O" as if it were an invalid Operation Field. However, the Operand Field is still processed correctly, and the generated object code is still the right code for the instruction.

The required operand must be an absolute expression with predefined terms. The binary value of the operand may specify any combination of the following machines:

Bit	Hex Value	Instruction Set
02	:04	LSI-2
01	:02	LSI-1
00	:01	ALPHA-16

The instruction subset common to all machines is always valid, and is equivalent to an explicit MACH value of binary 000.

The assembler initially sets the MACH value to binary 010. Each MACH value is retained until replaced by the next.

An appendix to this publication specifies the members of each machine instruction set.

Conditional Assembly Control (IFT/IFF/ENDC)

```
            IFT         Operand
            IFF         Operand
            ENDC        Comments
```

These directives specify whether a group of source statements is to be processed or discarded. Conditional assembly begins each time an IFT or IFF statement is encountered, and ends when the corresponding ENDC is found.

The required operand of an IF statement is an absolute expression with predefined terms. The operand is always analyzed for its Truth Value:

```
    0 means False
    1 means True
    Any other value means True
```

IFT means Assemble If True. All the statements bounded by an IFT and its corresponding ENDC are assembled if the operand of the IFT is True, and skipped otherwise.

IFF means Assemble If False. All the statements bounded by an IFF and its corresponding ENDC are assembled if the operand of the IFF is False, and skipped otherwise.

If the value of V is True, the LDA/LDX statements in the following example will be assembled, and the STA/STX statements will be discarded without being processed at all.

```
            IFT         V
            LDA         FLDA
            LDX         FLDX
            ENDC
*           *
            IFF         V
            STA         FLDA
            STX         FLDX
            ENDC
```

Conversely, if the value of V is False, the LDA/LDX statements will be skipped, and the STA/STX statements will be assembled.

Every IF must have corresponding ENDC somewhere below it. An IFT True or an IFF False with a missing ENDC will not affect the assembly, but will be flagged. An IFT False or an IFF True with no ENDC, however, will skip all the way to the END statement.

Define New Op Code ($class)

$class   Mnemonic   :hhhh

This directive communicates to the assembler the Mnemonic to be used for a new machine instruction (or a variant of an existing one), and specifies the object code to be generated by the new Mnemonic.

The directive consists of a Currency character in Column 1 of the source statement. This character is never used in Column 1 for any other purpose. The immediately following 1 or 2 columns contain the Class Number of a standard assembler Syntax Class.

The detailed operand requirements for each Syntax Class are described in another section. The machine level representations of the operands are described in the Appendix for each machine. The Syntax Classes and their most distinctive features are summarized in the following table.

Class Number	Words Generated	Machine Function	Operands Allowed	Indirect Mode	Indexed Mode	Other Mode
1	1	Word Reference	1	*	@	=
2	1	Byte Immediate	1			
3	1	Conditional Jump	2			
4	1	Single Register	1			
5	1	Register and Control	0			
6	1	Input/Output	2			
7	1	Double Register	1			
8	1	Byte Reference	1	*	@	
9	2	Double Register Arithmetic	1	*		
10	2	Stack Reference	1		@	+ or -

The $class directive must appear in the Source Program before any statements which generate object code.

OMEGA3 does not accept $7, $9, and $10.

The new Mnemonic consists of 1 to 4 alphanumeric characters, the first of which must be alphabetic. Embedded colons are permitted by the assembler, but are reserved for CA-supplied software.

The New Op Code Mnemonic may replace any existing Mnemonic for a machine instruction or a previously defined New Op Code. The new Mnemonic cannot replace a standard assembler directive.

The required operand is a 4-digit hexadecimal number. It specifies which bits in the first word of the generated object code are to be forced to 1's by the assembler. This bit pattern is called the Skeleton of the instruction.

The operands used with the New Op will determine the final appearance of the object code. Appendices C and D describe how the contents of certain bit fields are either calculated from the operand values, or set by various Address Mode specifications.

As examples of defining a New Op Code, some Skeletons built into the assembler for convenient coding of LSI-2 instructions will be reconstructed.

The following two statements are equivalent:

```
            JMP         $
            WAIT
```

WAIT has no operands, so it must be in Class 5. JMP $ is a Class 1 instruction, with one operand, and generates a fixed word of code, :F600. The New Op Code is thus defined by:

```
$5          WAIT        :F600
```

The following two statements are equivalent:

```
            JMP         *NAME
            RTN         NAME
```

Both RTN and JMP require exactly one Word Reference operand; both are in Class 1. The Skeleton for JMP, flagged Indirect, is :F100. The definition of RTN, therefore, is:

```
$1          RTN         :F100
```

Finally, consider the following sequence, which might be used to transfer control in a uniform way to external subroutines:

```
            JST         *$+1
            DATA        SUBR
```

Suppose a New Op Code were desired, so the two lines could always be replaced by:

```
            DO          SUBR
```

DO has exactly one operand. The generated object code must be two words long, and must contain the address of the operand in the second word. Syntax Class 9 fits the intended source statement format.

The existing machine instructions in Class 9 are used for Double Register Arithmetic functions, but the machine level functions of a New Op need not be related to the functions of any other instruction in the same class.

The Skeleton for JST *$+1 is the fixed word :FB00. The New Op Code definition is:

```
$9          DO          :FB00
```

Subroutine Structure Mnemonics (CALL/ENT/RTN)

[Label]	CALL	Name
Name	ENT	Comments
[Label]	RTN	Name

These Mnemonics provide a uniform way to communicate with a closed subroutine. They are not directives, and may be replaced by other definitions.

CALL is used as an executable operation, equivalent to the machine instruction JST. It performs two functions:

1. Store the Return Link -- the address of the next instruction after the CALL -- at the effective memory location of the operand.

2. Transfer control to the first word after the stored Return Link.

The operand of a CALL may be any operand valid for a Word Reference instruction. Ordinarily, the name of an ENT is used. If the name has been declared External, an implicit indirect reference thru a Literal Pool or thru Scratchpad might be used. An explicit indirect reference thru a REF is another possibility.

ENT is used as the destination of a CALL and of a RTN. The generated machine code is not intended for inline execution; it is simply a word of storage reserved for the Return Link by assembling a HLT instruction. The first executable instruction in the subroutine is coded immediately following the ENT. The ENT name may be local to the program, or declared a Primary or Secondary Entry as needed.

RTN is used to return to the calling program. It is equivalent to JMP *Name, and will perform an unconditional transfer of control indirectly thru the Return Link. The operand of a RTN is therefore identical to the name of the corresponding ENT.

Section 5

SYMBOL AND DATA DEFINITION

The directives in this section are used to generate non-executable object code, and to define symbols as the names of locations or values within the Source Program.

The directives DATA, TEXT, and BAC correspond to the three data types which are meaningful at machine level:

Word Value, or Word Address
Byte Value, or Character String
Byte Address

The directives EQU and SET use terms or expressions to assign values to symbols. EQU fixes a symbolic value for the entire Source Program; SET allows the symbolic value to vary.

Data Definition (DATA)

```
[Label]     DATA        [*]Operand[,[*]Operand]...
```

The DATA directive allocates storage for a number of words, and specifies the contents of each word.

The optional label is the location of the first allocated word.

The DATA statement requires at least one operand. Each operand may be any absolute or relocatable expression. The contents of a generated word may be flagged as an Indirect Address, by prefixing the corresponding operand expression with an asterisk.

An External symbol may be used as an operand. No Indirect Address prefix is accepted for an External reference.

The operands may be supplied in an arbitrary mixture of absolute, relocatable, direct, and indirect values. Reference to the Location Counter -- the symbol $ -- within an operand expression is taken to be the location of the specific word generated by that operand.

```
A           DATA        0,-132,'LP',*:FF,32767,$-A
*
R           DATA        $,R,*R+3,*$
*
X           DATA        SUB1,SUB2
```

Statement A generates 6 words, each containing an absolute value. The nominal location and the 16-bit contents of each word appear on a separate line of the assembly listing.

Statement R generates 4 words of relocatable data. The first 2 words contain the same value -- the relocatable address of R -- and the last 2 words both contain the indirect address of R+3.

If the names SUB1 and SUB2 are declared to be External in the Source Program, then the 2 words generated by statement X will not show any values on the listing. Later processing of the Object Program by the loader will insert correct values in the low-order 15 bits of each word.

Equate Symbol Value (EQU)

```
Name         EQU         Operand
```

This directive is used to define a symbol and its value without allocating any storage to the symbol. EQU statements may be used anywhere in the Source Program, but they are particularly useful in defining symbols which will be used extensively as terms in expressions.

The name of the symbol to be defined is specified in the required Label Field, and must be unique among all the symbols in the Source Program.

The EQU statement requires exactly one operand. The operand may be any absolute or relocatable expression, except that reference to an External is not allowed. Forward references are acceptable, but a directive which requires predefined operands (such as an ORG or an IF) cannot use a symbolic term defined by an EQU with forward references.

This example uses EQU to establish the destination of a jump without attaching a label to a line of executable code. This technique facilitates modification of the Source Program.

```
            JMP         DEST
*           *
*           *
DEST        EQU         $
```

The size of a table may be assigned a symbol this way:

```
TAB         DATA        0,2,4,6,8
TABSZE      EQU         $-TAB
```

An arbitrary ASCII character, especially a non-printable one, may be given a symbolic name as a coding convenience:

```
CR          EQU         :8D
*           *
*           *
            CAI         CR
```

Set Variable Value (SET)

```
Name          SET          Operand
```

This directive is used to define or to redefine the value of a symbol. SET statements may be used anywhere in the Source Program, but they are particularly useful in the control of conditional assembly.

The name of the symbol, or SET Variable, to be affected is specified in the required Label Field. A SET Variable name is unusual in this respect: it may be used in the Label Field of more than one source statement without being rejected as a multiple definition. On the contrary, a SET Variable has exactly one definition at any given point in the Source Program, but that definition is replaced completely by another SET for the same variable, even if the new SET has an invalid operand.

The name of a SET Variable must not appear in the Label Field of any type of statement except a SET statement; such an appearance would constitute multiple definition.

The SET statement requires exactly one operand. The operand may be any absolute or relocatable expression, except that reference to an External is not allowed. Forward references are acceptable, but a directive which requires predefined operands (such as an ORG or an IF) cannot use a symbolic term defined by a SET with forward references.

As an example of using a SET Variable to control conditional assembly, suppose that special debugging code -- perhaps a coded halt -- is scattered throughout the Source Program, and is always surrounded by an IFT/ENDC pair:

```
              IFT          TEST
              STOP         :77
              ENDC         TEST
```

To determine whether or not a specific part of the program would be assembled with an embedded STOP, either of these statements could be inserted into the Source Program in as many different places as needed:

```
TEST          SET          0            NO DEBUGGING STOPS
TEST          SET          1            INCLUDE DEBUGGING STOPS
```

SET Variables are sometimes useful when a particular coding technique -- for example, heavy use of backward jumps to nearby labels -- adds too many entries to the Symbol Table, leaving insufficient room for accumulated Literals. In the following code, each Jump instruction has the same operand, but the value of the operand, and therefore the assembled machine code, corresponds to the closest preceding SET for the symbol BACK. Observe that forward Jumps cannot be coded with SET Variables.

```
BACK          SET          $
*             *
*             *
              JAG          BACK
*             *
BACK          SET          $
*             *
*             *
              JXZ          BACK
```

Reserve Storage (RES)

```
[Label]     RES        Count [,Value]
```

The RES directive allocates storage for a number of words. It may also be used to fill all of the allocated words with a uniform value.

The optional label is the location of the first allocated word. The required Count specifies the number of words to be allocated. The Count must be an absolute expression with predefined terms. The value of the expression may be zero only if no Value is supplied. The following two statements are equivalent:

```
TAG         RES        0
TAG         EQU        $
```

The optional Value operand specifies the uniform contents of every allocated word. The Value must be an absolute expression. Any combination of terms may be used, except that reference to an External is not allowed. The following RES statement is equivalent to the entire series of DATA statements shown.

```
TAG         RES        3,:FF
*           *
TAG         DATA       :FF
            DATA       :FF
            DATA       :FF
```

Note that a repeated DATA statement may have a relocatable expression as its operand, but that a RES is more convenient to code if the desired storage contents represent an absolute value.

If a Value field is not supplied, neither the assembler nor the loader will alter the reserved locations. This facilitates either a source overlay, in which the RES locations are part of a backward ORG, or an object overlay, in which the loader does not disturb existing values in memory while loading object code allocated by a RES with no Value specification.

Text Definition (TEXT)

```
[Label]     TEXT        'String'
```

The TEXT directive allocates storage for a number of words, and specifies the contents of these words as a single ASCII character string.

The optional label is the location of the first word of allocated storage, which always starts at the first available word location, even though the storage is filled with byte values.

The required operand is an arbitrary string of ASCII characters, including any desired blanks and non-printable characters. The string must be delimited with a preceding and a following Single Quote or Apostrophe character.

If a character in the generated string must itself be a Single Quote, it is represented by two successive Single Quotes in two columns of the source statement. This should not be confused with a single character called Double Quote, which has no special significance in a TEXT string, and is therefore useful in punctuating assembled messages.

The characters in the TEXT string each represent one 8-bit byte, and are packed into successive words until the string is exhausted. The assembler will fill the low-order bits of the last word, if necessary, with :A0, an ASCII blank.

```
TAG         TEXT        'THIS IS A SIMPLE MESSAGE'
WHAT        TEXT        ' '''''  COMMENT
```

The contents of the two words starting at WHAT will be blank/quote/quote/blank:

```
            :A0A7
            :A7A0
```

Each word generated by a TEXT statement appears on a new line of the assembly listing.

Byte Address Constant (BAC)

```
[Label]     BAC         Operand [,Operand] ...
```

The BAC directive allocates storage for a number of words, and specifies that the contents of each word is the address of a byte location.

The optional label is the location of the first allocated word.

The BAC statement requires at least one operand. Each operand may be any absolute or relocatable expression, except that reference to an External is not allowed.

Each self-defining term in a BAC operand is used without change during evaluation of the operand expression. For example,

```
            BAC         :05
```

references the fifth byte of memory, and the word generated for the BAC contains :0005.

Each symbolic term, even if it was defined by a SET or EQU to a self-defining term, is always considered a word value, and is multiplied by 2 before evaluation of the operand expression.

```
Q           EQU         7
FLD         TEXT        'WXYZ'
            BAC         Q
            BAC         FLD
```

Each of these BAC operands is a symbolic term. The first references the seventh word of memory, which is the fourteenth byte; the generated word contains :000E. Similarly, the value of FLD, whether absolute or relocatable, must be doubled to produce a byte value.

An odd-numbered byte -- that is, the low-order byte within a given word-- may be referenced by using an odd self-defining term in the operand expression:

```
            BAC         FLD+1,FLD+3
```

This statement will generate two words, containing the byte addresses of the characters "X" and "Z" in the assembled text.

Section 6

LOCATION CONTROL

The directives in this section specify a new value for the Location Counter -- the nominal location of the object code -- and for the Load Attribute -- Absolute or Relocatable.

The segment of code following each directive is called the range of the directive. A range terminates with the next Location Control directive, or with an END statement.

Within a given range, the symbol $ (which represents the current value of the Location Counter), or a symbol defined as the Label of a storage allocation or a machine instruction, acquires the Load Attribute of that range. Similarly, a Label defined by a simple reference to $ has the same Load Attribute as $, and the same as the current range:

```
TAG          EQU          $
TAG          SET          $
```

A label defined with an EQU or a SET to a multi-term expression, however, acquires the Load Attribute of the evaluated expression, regardless of the current range.

Absolute Object Code (ABS)

```
            ABS         Operand
```

This directive sets the Load Attribute to Absolute, and the Location Counter to the value of the operand. The result is a segment of object code which is loaded to begin at a fixed location in memory.

The required operand is an absolute expression with predefined terms. The expression must have a positive (or zero) value.

The source statements shown here are the first few lines of the assembler itself. They begin at location :0000 Absolute, so the generated object code will always occupy the first 6 words of memory.

```
            ABS         0
            STOP        :99         POWER UP INTERRUPT
            JMP         *NXTP       TO EDITOR
CORLM       RES         1           CALCULATED HIGH MEMORY LIMIT
MCHDEF      DATA        2           DEFAULT VALUE OF MACH
LINES       DATA        -53         LINES PER PAGE - 13 (NEGATIVE)
CHARS       DATA        -72         CHARACTERS PER LINE (NEGATIVE)
```

Relocatable Object Code (REL)

```
            REL         Operand
```

This directive sets the Load Attribute to Relocatable, and the Location Counter to the value of the operand. The result is a segment of code which is loaded to begin at a location calculated as the sum of:

1. The REL operand value, plus
2. The Relocation Bias parameter supplied to the loader, plus
3. The next available location in memory, as REL code accumulates in the successive Object Programs being loaded together.

The Location column on the assembly listing contains the nominal location for each word in a Relocatable range -- that is, relative to the REL operand.

The required operand is an expression with predefined terms. The Load Attribute of the evaluated expression may be either Absolute or Relocatable.

For almost all applications, the following technique is appropriate for the main program, and for each separately assembled subprogram.

```
            TITL        PROGRAM XXX -- VERSION VV
            NAM         XXX AND OTHERS AS NEEDED
            REL         0
*           *
*           *           REST OF PROGRAM
*           *
            END         (TRANSFER ADDRESS IF NEEDED)
```

This technique defers until load time the question of where in memory the program will be executed. If fixed absolute memory locations are desired later, the Object Program can be loaded, then punched out with the Binary Dump utility.

Origin of Object Code (ORG)

```
            ORG         Operand
```

This directive sets the Location Counter to the value of the operand. It does not alter the current Load Attribute. The result is a segment of code which is loaded to begin at a location discontinuous from the previous segment, but with the same bias applied.

The Location column on the assembly listing reflects the discontinuity in nominal location caused by an ORG.

The required operand is an expression with predefined terms. In particular, no term may be a forward reference -- this error often occurs when pieces of a Source Program are rearranged. The Load Attribute of the expression must be consistent with the ABS or REL range into which the ORG itself falls.

A forward ORG is equivalent to a RES with no second operand -- no specification of a value to be filled in. This sequence reserves two card input buffers:

```
CARDSZ      EQU         80
BUFF1       EQU         $
            ORG         BUFF1+CARDSZ
BUFF2       EQU         $
            ORG         BUFF2+CARDSZ
REST        EQU         $
```

A backward ORG is used to overlay, at load time, an area previously defined. The same location may be ORG'd back to as many times as needed. The last value assembled will be the last one inserted by the loader.

The following sequence generates 256 consecutive words filled with binary 1's; then ORGs back to the 64th word and clears it; then ORGs forward past the end of the table, so unrelated data can follow.

```
TABLE       RES         256,:FFFF
*
            ORG         TABLE+63
TABZRO      DATA        0
            ORG         TABLE+256
*
MORE        DATA        2,4,8,16
```

A common coding error, and a difficult error to detect, is a backward ORG without a later forward ORG, or without enough code-generating statements to bring the Location Counter forward as far as intended. If the last ORG were omitted in the preceding example, all of TABLE beyond TABZRO would be destroyed at load time by the data starting at MORE.

Section 7

OBJECT PROGRAM LINKAGE

The directives in this section are used to establish communication between separate Object Programs. They generate records on the Punch Output which contain distinctive Loader Type Codes, meaningful to LAMBDA2 and LAMBDA3.

An Object Program which contains a Loader Type Code corresponding to any directive in this section cannot be loaded with BLD2, BLD3, or Autoload. However, the Object Program can be processed thru LAMBDA and BDP (or thru OS:LNK) to produce a new tape acceptable to BLD or Autoload.

Entry Declaration (NAM/SNAM)

```
NAM         Name [,Name]...

SNAM        Name [,Name]...
```

These directives are used to declare that certain names are to be made available to the loader for possible matching against unresolved Externals in other programs. Each name must be defined somewhere within the assembly, either as a relocatable or as an absolute symbol. The name may be defined with an EQU statement, but it must not be a SET Variable.

NAM declares each name to be a Primary Entry. A Primary Entry which matches an unresolved Primary External will force selection of the program which contains the Primary Entry. A Primary Entry may also be resolved against a matching Secondary External, once both programs have already been selected.

SNAM declares each name to be a Secondary Entry. A Secondary Entry will never force selection, but it will be available for matching against an unresolved Primary or Secondary External, once both programs have already been selected.

All the Primary Entries in an Object Program must be presented to the loader before the Object Program is processed. Therefore, the assembler imposes a restriction upon the placement of NAM statements (but not SNAM statements) in a Source Program -- they must appear before any machine instructions, and before any directive which generates object output, including EXTR, LOAD, REL, and ABS. The recommended placement for NAM statements is at the very beginning of the Source Program, preceded only by TITL and Comment Line statements.

External Declaration (EXTR)

```
          EXTR        Name[,Name]...
```

This directive is used to declare that certain names may eventually appear as Entries in other programs selected during load processing. Each name must be acceptable as a label, but must not be defined anywhere in the assembly.

EXTR declares each name to be a Primary External. An unresolved Primary External which matches a Primary Entry will force selection of the program which contains the Primary Entry. An unresolved Primary External may also be resolved against a matching Secondary Entry, once the program containing the Secondary Entry has already been selected.

The mere appearance of a name in an EXTR statement is not sufficient to create an unresolved External. The name must actually be referenced somewhere in the assembly before it is considered unresolved.

Because the value of an External Name is not available to the assembler, a symbol declared in an EXTR statement can be used only in certain restricted contexts:

Word Reference machine instruction (Class 1)
Double Register Arithmetic machine instruction (Class 9)
Stack Reference machine instruction (Class 10)
DATA statement
SPAD statement

An External Name cannot be used as a term in a complex expression, but it can be used in isolation in a context where an expression would be acceptable. Neither a Unary Plus nor a Unary Minus can be prefixed, nor is an asterisk (indicating an Indirect Address) valid as a prefix.

Here are examples of all the contexts in which an External Name can appear.

```
          EXTR        SUBR        DECLARATION
*         *
          LDA         SUBR        CLASS 1, DIRECT
          LDA         =SUBR       CLASS 1, LITERAL
          LDA         @SUBR       CLASS 1, DIRECT INDEXED
          MPY         SUBR        CLASS 9
          XORS        SUBR        CLASS 10
          DATA        SUBR        DATA OPERAND
          SPAD        SUBR        SPAD NAME
```

Demand Load (LOAD)

```
          LOAD          Name [,Name] ...
```

This directive is used to create unresolved Primary Externals. Typically, each name is resolved against a matching Primary or Secondary Entry by the loader.

A name declared in an EXTR is a Primary External, but is not considered unresolved unless the name is actually referenced somewhere in the assembly. No such reference is needed for a LOAD name.

A name declared in a REF is an unresolved Primary External, but each REF allocates a word of storage, and a name cannot appear in more than one REF in an assembly. No storage is consumed by a LOAD, and a name can appear in any number of LOAD statements.

Suppose these two subprograms are placed on an Object Program Library:

```
*           SUB         AC
            NAM         XA
            REL         0
            SNAM        XC
XA          EQU         $
XC          EQU         $
*           *
*           *
            END


*           SUB         B
            NAM         XB
            REL         0
XB          EQU         $
*           *
*           *
            END
```

This main program is assembled, and submitted to the loader first:

```
*           MAIN
            REL         0
            LOAD        XL
*           *
*           *
XA          SREF
XB          SREF
XC          SREF
            END
```

One, and only one, of these two segments is submitted to the loader after MAIN, and before AC and B:

```
*           XL VERSION  A        ||     *             XL VERSION  B
            NAM         XL       ||                   NAM         XL
XL          RES         0        ||     XL            RES         0
            LOAD        XA       ||                   LOAD        XB
            END                                       END
```

If XL Version A is used, MAIN is loaded with Subprogram AC. References to both XA and XC are resolved. References to XB are left unresolved.

If XL Version B is used, MAIN is loaded with Subprogram B. References to XB are resolved. References to both XA and XC are left unresolved.

Two points are of particular interest here:

1. MAIN has no use for XL itself. Except for the LOAD, no statement in MAIN even references XL. What MAIN wants is some combination of XA, XB, and XC.

2. XL occupies no storage at all. It is not really a subprogram, but a technique to control the loading process.

The use of a name in a LOAD statement does not constitute a definition of a symbol within the current Source Program. The same name could be defined as a Label (or as a SET Variable) within the current assembly, and would have no connection with the information in the LOAD statement. If such a Label were also declared in a NAM or SNAM statement, however, it would be available to the loader for possible matching against the name in the LOAD statement, just as if it appeared in some other Object Program.

Reserve Chain Link (CHAN)

```
[Label]      CHAN        [*]Identifier
```

This directive facilitates the creation of a type of data structure known as a "chain" or "linked list" or "threaded list." An example of chain structure and usage follows this description.

For each use of the CHAN directive, the assembler reserves one word of storage. The optional label is the location of this word, and may be used in any context as if it were the label of a RES directive.

The required operand, called the Identifier, consists of 1 to 6 alphanumeric characters, the first of which must be alphabetic. Embedded colons are permitted by the assembler, but should be reserved for CA-supplied software.

All CHAN directives having precisely the same Identifier contribute storage to one specific chain structure at load time, regardless of whether the directives appeared in one assembly or in several programs loaded together.

The use of a particular alphanumeric string as an Identifier does not constitute a definition of a symbol. The Identifier, as such, cannot appear in any statement other than a CHAN. In theory, the same string could be used as the label of a statement, and references to that label would be valid. In practice, using the same string both as a chain Identifier and as an ordinary label is confusing and inadvisable.

An optional asterisk may be prefixed to the Identifier. At load time, a high-order "1" bit will be set in the word reserved by the CHAN directive. The meaning attached to this bit is defined by the user's own chain-processing routine.

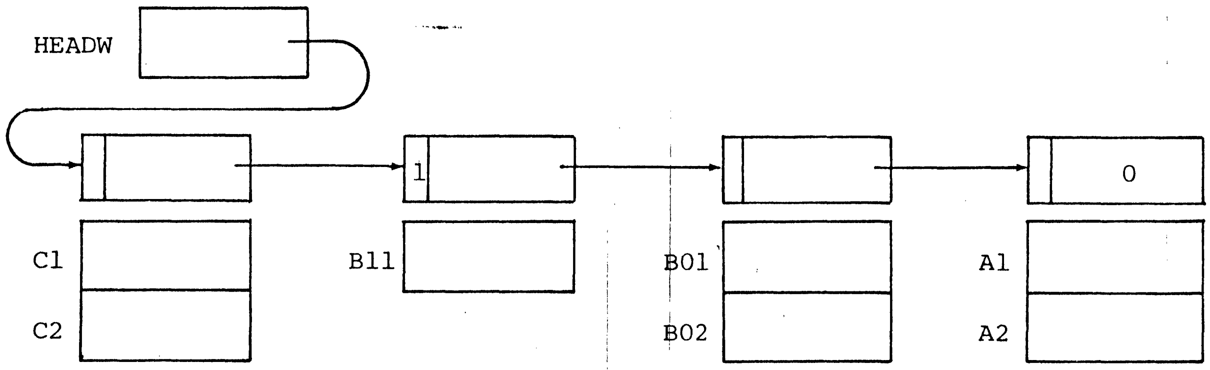
The words which belong to a specific chain -- its links -- are filled in at load time. It must be understood that the mere appearance of a chain Identifier is not sufficient reason for a given program to be selected by the loader; which programs are selected, and which are not, is governed solely by resolution of External references, to which the CHAN directive contributes nothing.

When a word reserved by the CHAN directive is encountered, its high-order bit is set according to the user's specification, and the remaining 15 bits are made a direct storage address. For a particular chain, the very first link processed is set to :0000 or :8000. This zeroed link is called the tail of the chain.

The second link in each chain contains the storage address of the tail; the third link contains the address of the second link; and so on, until no links remain in the program. It is the responsibility of the program to know where the last link, or head of the chain, is located. This implies careful control over the order in which Object Programs, and the CHAN directives within them, are presented to the loader.

Example of Chain Structure and Usage

This chain is created by the CHAN and DATA directives shown:



```
*     PROGRAM   A      *     PROGRAM   B      *     PROGRAM   C
      CHAN      W            CHAN      W            CHAN      W
A1    DATA      0      B01   DATA      0      C1    DATA      0
A2    DATA      0      B02   DATA      0      C2    DATA      0
                       *     STORAGE
                       *     UNRELATED
                       *     TO CHAIN W
                             CHAN      *W
                       B11   DATA      0
```

The chain is processed by this program, which must be loaded last:

```
AHDW      DATA      HEADW
HEADW     CHAN      W         HEAD OF CHAIN W
*
          LDX       AHDW      INITIALIZE POINTER
LOOPW     LDX       @0        X NOW CONTAINS A LINK
          LLX       1         ELIMINATE POSSIBLE
          LRX       1           FLAG FROM LINK WORD
          JXZ       ENDW      IF LINK = 0, NO MORE PROCESSING
*
*         PROCESS DATA AT @1 AND @2 HERE
*         FLAG MAY BE CHECKED BY REFERENCE TO @0
*
          JMP       LOOPW
ENDW      EQU       $
```

External Reference Constant (REF/SREF)

```
Name         REF          Comments

Name         SREF         Comments
```

These directives are used to declare that certain names are to be considered both internal and external references, so that explicit linkage to another program may be used.

Within the assembly, the name is recognized as the label of a single word of storage, which is reserved just as if the statement had used RES 1 rather than REF or SREF. The name, therefore, must not appear in the label field of any other statement in the assembly.

At load time, the name is presented to the loader as an unresolved External. If a matching Entry becomes available in another Object Program, the word reserved by the REF or SREF is filled in with the direct address of the Entry.

The statement sequence shown here involves an implicit indirect link thru a word in a Literal Pool or -- if no such word is available within addressing range -- a word in Scratchpad:

```
             EXTR         SUBR
             JST          SUBR
```

The following sequence allows the programmer to control explicitly the storage allocation for the link, or even to build a table of External pointers:

```
SUBR         REF
             JST          *SUBR
```

A REF statement creates an unresolved Primary External. An SREF statement creates an unresolved Secondary External.

Section 8

LITERALS

A Literal is a word of storage, allocated for the operand of a Word Reference or Byte Reference machine instruction. Unlike a word allocated by a DATA statement, the exact location of a Literal is chosen not by the programmer, but by the assembler itself. In certain cases, the fact that a Literal was required is unknown to the programmer until the assembly listing is available for inspection.

A collection of Literals, grouped together in one area of memory, is called a Literal Pool. The programmer can exercise some control over the location and size of a Literal Pool, but again the assembler makes some of the decisions by itself.

Two coding techniques always generate Literals. One is an Explicit Literal operand -- that is, the source statement operand expression is prefixed by an = sign. Rather than writing:

```
            ADD         K1000
```

and remembering several pages later to include:

```
K1000       DATA        1000
```

the programmer writes:

```
            ADD         =1000
```

and lets the assembler allocate the storage, fill in the value, and adjust the machine instruction address.

The other technique which predictably needs a Literal is a reference to a name already declared External, and thus beyond any possible Direct Relative Addressing Range. Typically, a subroutine call is involved:

```
            EXTR        SUBR
*           *
            JST         SUBR
```

The assembler makes the machine instruction indirect, and allocates a word in a Literal Pool for the subroutine address. The result is the same as if the programmer had written something like:

```
            JST         *XSUBR
*           *
            EXTR        SUBR
XSUBR       DATA        SUBR
```

A related coding technique may or may not generate a Literal. In this case, backward reference is made to a location which has already been defined. If the assembler calculates that the location falls too far back for Direct Relative Addressing, the machine instruction is made indirect, and an intermediate link is created in a Literal Pool.

```
PARTA       EQU         $
*           *
*           *
PARTB       EQU         $
*           *
*           *
CYCLE       JMP         PARTA
```

If the code in PARTA and PARTB is still under development, the distance between CYCLE and PARTA may fluctuate in and out of JMP range with each re-assembly. This fact is ordinarily of no concern to the programmer, because the assembler will decide for itself which Addressing Mode is needed.

The need for each Literal arises within a segment of executable instructions. This is exactly where the assembler can not allocate storage for the Literal, which is a word of data. Instead, Literals accumulate until the programmer designates an appropriate location for them with an LPOOL directive.

This process leads to the fourth, and final, coding sequence which can generate a Literal. Again, the assembler's helpfulness in the calculation of Relative Addressing Ranges is involved.

```
LOOP        LDA         FLDB
            LDX         =1000
*           *
            JMP         LOOP
*           *
            LPOOL
FLDA        DATA        0,2,4,6,8,10
FLDB        DATA        0
*           *
*           *
```

When the assembler first processes the source statement labelled LOOP, the reference to FLDB is still undefined. It is not an External, but it is a forward reference, and may or may not prove to be out of range. The assembler provisionally decides that a Literal would guarantee access to FLDB, makes the LDA indirect, and adds the Literal to the current accumulation. The Explicit Literal in the LDX also joins the accumulation.

The programmer finishes writing executable code, and begins some DATA statements. But first, to provide for the Explicit Literals in the last piece of code, and perhaps some other accumulated Literals, LPOOL is inserted. Among the words immediately allocated under the LPOOL, the assembler includes one for the reference to FLDB, another for =1000.

Now the assembler finds out where FLDB is, in relation to LOOP. If FLDB is out of range, the Literal Pool entry really was needed, and the indirection already set in the LDA is the only way to access FLDB.

Suppose, however, that FLDB turns out to be within range of the LDA. The instruction is made direct to save execution time. The Literal Pool word, which would have been a pointer to FLDB, is left unfilled.

The allocated storage remains in the program. Removing the allocation would involve reassembly of the entire Source Program.

Literals take up storage. Techniques which generate Literals may use the storage efficiently, and they may not. Only the programmer, not the assembler, can make that decision.

To summarize, these techniques may generate Literals for Word Reference or Byte Reference instructions:

1. Prefixing an operand with an = sign.
2. Reference to a location known to be External.
3. Backward reference to a location beyond Direct Relative Backward Addressing Range.
4. Forward reference to a location not defined before the next LPOOL statement.

Allocate Literal Pool (LPOOL)

```
[Label]     LPOOL     [Operand     [Comments]]
```

This directive informs the assembler that it may allocate storage for whatever Literals have been accumulated. The optional label is the location of the first allocated word.

No words are allocated if no Literals have been accumulated. Even the use of an Explicit Literal between one LPOOL and the next does not always require a new Literal Pool entry.

```
A           LDA         =1000
*           *
B           LDA         =500*2
*           *
L1          LPOOL
*           *
*           *
C           LDA         =4*250
*           *
L2          LPOOL
```

The Literal for =1000 in Literal Pool L1, originally created for instruction A, is shared with instruction B -- the assembler can see that the same value is involved, even if the source expression looks different. Furthermore, when C is processed, the assembler checks for a matching value in all the Pools within backward range before it assumes that a new value will be needed in a forward Pool. This can result in very efficient sharing of Literal Pool allocations, if the programmer places LPOOL statements judiciously.

For C to share the Literal created for A, the starting location of the Pool at L1 must be within the Relative Backward Addressing Range of C. It is not sufficient that the word allocated for the =1000 be within range; the entire Pool must be close enough.

If L1 is not within range of C, a new Literal also containing =4*250 (that is, =1000) becomes part of the forward Pool at L2. The new value is available for sharing with instructions beyond L2 but within range of it.

The optional operand of an LPOOL statement is an absolute expression with predefined terms and a value greater than zero. It specifies the maximum number of words allowed in this Literal Pool, regardless of how many Literals have been accumulated. If more words are needed, the leftover Literals will be held for the next available Literal Pool.

The programmer should observe that no Comments may be used in an LPOOL statement which has no operand.

If an assembly contains at least one LPOOL statement, than all the Literals still accumulated when the END statement is reached are allocated just as if the END were immediately preceded by an LPOOL. A dummy statement of LPOOL 1 at the start of the assembly is sufficient to activate this provision for leftover Literals.

If an assembly contains no LPOOL statements at all, then no Literal Pools are ever generated. Instead, every instruction which would have used Relative Addressing into a nearby Literal Pool is set for Indirect Scratchpad Addressing. All of the Literals are converted into Scratchpad Literals, which are described in the next section of this manual.

Section 9

SCRATCHPAD LITERALS

A Scratchpad Literal is a word of storage allocated by the loader, and available to a Word Reference or Byte Reference instruction thru Scratchpad Addressing Mode. The need for a Scratchpad Literal is determined during the assembly process, and communicated from the assembler to the loader thru a distinctive Loader Type Code in the generated Object Program.

Two coding techniques result in Scratchpad Literals. The more common situation is that a Literal Pool Reference, either explicit or implicit, was used, but that no Literal Pool space was available within range of the instruction which involved the reference. This includes the extreme case of a Source Program which never uses an LPOOL at all, such as a program originally coded for CA-supplied assemblers lacking such a directive.

If at least one LPOOL statement is found in a Source Program, it is assumed that the programmer wanted to minimize or eliminate any requirement for Scratchpad Literals. Therefore, the assembler will attach a Warning Flag to every Class 1 or Class 8 instruction which needed a Scratchpad Literal only because no LPOOL was within Relative Addressing Range.

Certain ways of using instructions always need Scratchpad Literals, and will not be flagged. Specifically, a Word Reference or Byte Reference operand with the prefix @ -- which indicates Indexed Addressing -- will always be generated with a Scratchpad Literal for indirect linkage if the operand value is either:

1. Relocatable, or

2. Absolute, but higher than the machine limit for Direct Indexed Addressing (:3F for the 3/05, :FF for the other machines).

Even a combination of Literal Pool entries and Scratchpad Literals cannot guarantee that a Byte Reference instruction has access to every location in memory. The assembler rejects a Byte Reference instruction with Explicit Indirect Addressing if its operand (presumably the location of a Byte Address Constant) is not within Direct Addressing Range. Neither a Scratchpad link nor a Literal Pool word can be used to access the BAC, and thru it the actual data, because only one level of Indirect Addressing is available when the machine is in Byte Mode.

Scratchpad Literal Only (SPAD)

```
          SPAD        Name[,Name]...
```

This directive declares that certain names are to be excluded from ordinary Literal Pool allocation. If at least one term of the operand expression of a Word Reference or Byte Reference instruction is an SPAD name, and the assembler finds that a Literal is needed, then the Literal will go into the Scratchpad Literal Pool.

Each name may be local to the assembly, or it may be declared External, or it may never appear at all. An SPAD name may appear in a number of different SPAD statements. An SPAD statement only affects other statements after it, not before.

An SPAD name is usually declared because the programmer is using LPOOL directives, but anticipates that frequent references to a certain name would generate a considerable number of unshared words in many different Literal Pools. In this situation, a Scratchpad Literal is more conservative of storage, because the loader eliminates duplicate values before allocating the Scratchpad Literal Pool.

Section 10

INTERPRETATION OF THE ASSEMBLY LISTING

This section describes the information on the assembly listing. A sample listing follows the description.

Page headings have already been discussed under TITL. Two kinds of lines appear in the body of the listing, Error Lines and Statement Lines.

Error Lines

An Error Line starts with two asterisks and a blank. Various flags follow, each of which represents an error in the source statement on the immediately preceding line. The specific meaning of each flag is listed for ready reference at the end of this section.

At the very end of the listing, this message appears:

```
    yyyy ERRORS eeee
```

The number yyyy is the total number of lines with Error Flags. The number eeee is a chainback pointer. The last source statement which caused an Error Flag was statement eeee on the listing. The Error Line under that statement contains a chainback to the next-to-last statement which caused an Error Flag, and so on back to the first Error Flag, which is easily recognized by its lack of a chainback pointer.

Statement Lines

A Statement Line is divided into 7 uniform columns, separated by one or two blanks:

1. Line Number
2. Location
3. Value
4. Label Field
5. Operation Field
6. Operand Field
7. Comments Field

Line Number

This column identifies each source statement.

Location

The current value of the Location Counter appears in this column.

Value

The result of assembling each statement is shown here. If a machine instruction or a directive generates object code, each word appears on a new line, so the Location column can be updated. If a statement simply evaluates an expression, the final value appears as a 16-bit word.

The Value column also supplies information about Literals. For an LPOOL statement, the number of words allocated in the Literal Pool is given. For a reference to an entry in a Literal Pool, the location of the word is shown below the object code. For a Scratchpad Literal, the value passed to the loader -- that is, the operand expression value -- is shown.

Source Statement Fields

The remaining columns on the assembly listing contain the four fields of the original source statement, spread into uniform columns.

Symbol Table

The main assembly listing is followed, on a new page, by the names and values of all the Symbols and SET Variables in the Source Program. The names are alphabetized, and displayed 4 across. Each name is followed by its 16-bit value. To the left of a name, these flags may appear:

M	Multiple Definitions
U	Undefined Symbol
X	External or Entry

If LPOOL directives were used, the alphabetized entries will be preceded by messages of this form:

```
    LPOOL@  hhhh
```

That is, "Literal Pool at location hhhh." Every Literal Pool, including the implicit one before the END statement, will be identified in order of appearance.

Error Flags

A	Absolute expression was required, but operand here is Relocatable. Value of operand expression is not an acceptable value for this Mnemonic Destination of a Conditional Jump is out of range.
C	ENDC not paired with an IFT or IFF. IFT or IFF range still open when END was reached -- ENDC missing.
D	Operand reference to a symbol with multiple definitions.
E	Expression could not be evaluated -- value forced to :0000 Absolute.
L	Label Field unacceptable.
M	Multiple definition of a symbol.
O	Operation Field unacceptable -- processed as if HLT.
P	Pass 2 out of synch -- probable error in hardware or software.
R	Relocation Factor unacceptable -- value forced to :0000 Absolute.
S	Syntax error in operand expression.
T	Self-defining term too large -- value forced to :0000 Absolute.
U	Undefined symbol was referenced.
W	Warning -- this Word Reference or Byte Reference instruction needs a Scratchpad Literal. (This flag appears if a Source Program contains at least one LPOOL statement. The same warning appears if no LPOOL statements were used, but the hardware SENSE switch is ON during Pass 2 processing.)
OV	Overflow of an intermediate value beyond 16-bit maximum. Statement processing was unsuccessful because of Symbol Table overflow.

```
PAGE  0001          SAMPLE ASSEMBLY LISTING FOR OMEGA

0002                ***************************************************
0003                *                SECTION 11
0004                *
0005                *              SAMPLE ASSEMBLY LISTING
0006                *
0007                ***************************************************
0008                *
0009                       NAM    MAIN
0010  0000                 REL    0
0011                *
0012        0000    MAIN   EQU    $
0013                *
0014        0002    ABS    EQU    +2          ABSOLUTE
0015                *
0016  0000  8802           ADD    ABS
0017  0001  8AA5           ADD    =ABS
            0027
0018  0002  0B02           AAI    ABS
0019  0003  1200           JAG    ABS
**  A
0020  0004  4513           AIB    ABS,3
0021  0005  8804           ADDB   ABS
0022                *
0023  0006  0002           DATA   ABS,ABS+3,*ABS
      0007  0005
      0008  8002
0024  0009  0004           BAC    ABS,ABS+3
      000A  0007
0025        0002    SETVAR SET    ABS
0026  000B  A29B           LDX    =SETVAR
            0027
0027  000C  8E9B           SUB    =ABS+7-ABS
            0028
0028                *
0029        1234    ABSBIG EQU    :1234       ABSOLUTE BEYOND SCRATCHPAD
0030                *
0031  000D  8B9B           ADD    ABSBIG
            0029
0032  000E  8A9A           ADD    =ABSBIG
            0029
0033  000F  0B00           AAI    ABSBIG
**  A       0019
0034  0010  1200           JAG    ABSBIG
**  A       0033
0035  0011  4500           AIB    ABSBIG,3
**  A       0034
0036  0012  8B97           ADDB   ABSBIG
            002A
0037                *
0038  0013  1234           DATA   ABSBIG,ABSBIG+3,*ABSBIG
      0014  1237
      0015  9234
0039  0016  2468           BAC    ABSBIG,ABSBIG+3
```

```
PAGE   0002

       0017   2468
0040          1234   SETVAR  SET    ABSBIG
0041   0018   A290           LDX    =SETVAR
              0029
0042   0019   8E8F           SUB    =ABSBIG+7-ABSBIG
              0028
0043                 *
0044          FFFE   NABS    EQU    -2          NEGATIVE ABSOLUTE
0045                 *
0046   001A   8800           ADD    NABS
** E          0035
0047   001B   8A8F           ADD    =NABS
              002B
0048   001C   0800           AAI    NABS
** A          0046
0049   001D   1200           JAG    NABS
** A          0048
0050   001E   4500           AIB    NABS
** A          0049
0051   001F   888C           ADDB   NABS
              002C
0052                 *
0053   0020   FFFE           DATA   NABS,NABS+3,*NABS
       0021   0001
       0022   FFFF
0054   0023   FFFC           BAC    NABS,NABS+3
       0024   FFFF
0055          FFFF   SETVAR  SET    NABS
0056   0025   A285           LDX    =SETVAR
              002B
0057   0026   8E81           SUB    =NABS+7-NABS
              0028
0058                 *
0059                 *
0060          0006   LP1     LPOOL
       0027   0002
       0028   0007
       0029   1234
       002A   2468
       002B   FFFF
       002C   FFFC
0061                 *
0062          0002   REL     EQU    MAIN+2   RELOCATABLE
0063                 *
0064   002D   8A54           ADD    REL
0065   002E   8A98           ADD    =REL
              0047
0066   002F   0800           AAI    REL
** A          0050
0067   0030   1211           JAG    REL
0068   0031   4500           AIB    REL,3
** A          0066
0069   0032   8A1E           ADDB   REL
```

```
0070                    *
0071   0033  0002             DATA   REL,REL+3,*REL
       0034  0005
       0035  8002
0072   0036  0004             BAC    REL,REL+3
       0037  0007
0073         0002       SETVAR SET    REL
0074   0038  A28F             LDX    =SETVAR
             0047
0075   0039  8C07             SUB    REL+7-REL
0076                    *
0077                          EXTR   SUBR       EXTERNAL
0078                    *
0079   003A  888D             ADD    SUBR
             0048
0080   003B  8A8C             ADD    =SUBR
             0048
0081   003C  0800             AAI    SUBR
** E         0068
0082   003D  1200             JAG    SUBR
** F         0081
0083   003E  4500             AIB    SUBR,3
** A         0082
0084   003F  8800             ADDB   SUBR
** E         0083
0085                    *
0086   0040                   DATA   SUBR,SUBR+3,*SUBR
       0041
** E         0084
       0042  8000
** E         0086
0087   0043  0000             BAC    SUBR,SUBR+3
** F         0086
       0044  0000
** E         0087
0088         0000       SETVAR SET    SUBR
** E         0087
0089   0045  A000             LDX    =SETVAR
** U         0088
0090   0046  8C00             SUB    SUBR+7-SUBR
** E         0089
0091                    *
0092                    *
0093         0003       LP2    LPOOL
       0047  0002
       0048  0000
       0049
0094                    *
0095         4567       RELFAR EQU    MAIN+:4567 RELOCATABLE OUT OF RANGE
0096                    *
0097   004A  8900             ADD    RELFAR
             4567
** A         0090
```

```
0098   004B   8800               ADD     =RELFAR
              4567
** A          0097
0099   004C   0800               AAI     RELFAR
** A          0098
0100   004D   1200               JAG     RELFAR
** A          0099
0101   004E   4500               AIB     RELFAR,3
** A          0100
0102   004F   8900               ADDB    RELFAR
              8ACE
** A          0101
0103                    *
0104   0050   4567               DATA    RELFAR,RELFAR+3,*RELFAR
       0051   456A
       0052   C567
0105   0053   8ACF               BAC     RELFAR,RELFAR+3
       0054   8AD1
0106          4567      SETVAR   SET     RELFAR
0107   0055   A000               LDX     =SETVAR
              4567
** A          0102
0108   0056   8E51               SUB     =ABS+7-ABS
              0028
0109                    *
0110                    *
0111                    * NEXT STATEMENT IS A NEW PAGE DIRECTIVE (.)
```

```
                                        CONDITIONAL ASSEMBLY DEMONSTRATION
0113                  *
0114                  *
0115          0001    TRUE     SET     1
0116          0000    FALSE    SET     0
0117                  *
0118                  * TAKE 7 SOURCE STATEMENTS LIKE THIS --
0119                  *                          IFT      TV
0120                  *                          STOP     :77
0121                  *                          ENDC
0122                  *                          NOP
0123                  *                          IFF      TV
0124                  *                          STOP     :88
0125                  *                          ENDC
0126                  *
0127          0001    TV       SET     TRUE
0128                  * FIRST, WITH TV = TRUE
0129                  *
0130          0001             IFT     TV
0131    0057  3C77             STOP    :77
0132                           ENDC
0133    0058  0000             NOP
0137                  *
0138          0000    TV       SET     FALSE
0139                  * NOW, THE SAME 7 STATEMENTS WITH TV = FALSE
0140                  *
0144    0059  0000             NOP
0145          0000             IFF     TV
0146    005A  3C88             STOP    :88
0147                           ENDC
0148                  *
0149                  * NOTE THE JUMP IN THE LINE NUMBER
0150                  * WHEN SOMETHING IS SKIPPED
```

```
PAGE  0006

0152                  *  THE NEXT STATEMENT WILL LEAVE NO LITERAL POOL
0153                  *  WITHIN FORWARD RANGE OF THE STATEMENTS
0154                  *  FOR 'RELFAR' -- BUT SOME OF THOSE STATEMENTS
0155                  *  WILL BE ABLE TO USE EXISTING VALUES IN POOL LP2
0156                  *
0157  015B                   ORG     $+:100
0158                  *
0159                  *
0160  015B  8800             ADD     0
0161  015C  8A8C             ADD     =0
            0169
0162  015D  0B00             AAI     0
0163  015E  1200             JAG     0
** A        0107
0164  015F  4503             AIB     0,3
0165  0160  8800             ADDB    0
0166                  *
0167  0161  0000             DATA    0,0+3,*0
      0162  0003
      0163  8000
0168  0164  0000             BAC     0,0+3
      0165  0003
0169        0000      SETVAR SET     0
0170  0166  A282             LDX     =SETVAR
            0169
0171  0167  8E82             SUB     =0+7-0
            016A
0172                  *
0173                  *
0174  0168  016D             DATA    ENDTAG   WORD AFTER END OF THIS PROGRAM
0175                  *
0176                  *  HERE COMES AN IMPLICIT LPOOL BEFORE END
            0004
      0169  0000
      016A  0007
      016B
      016C
0177        0000      ENDTAG END     MAIN
0029 ERRORS 0163
```

```
    LPOOL@   0027          LPOOL@   0047          LPOOL@   0169          ABSBIG   1234
    ABS      0002          ENDTAG   016D          FALSE    0000          LP1      0027
    LP2      0047     X    MAIN     0000          NABS     FFFE          RELFAR   4567
    REL      0002          SETVAR   0000     X    SUBR     0000          TRUE     0001
    TV       0000
```

Section 12

EDITING AND ASSEMBLING A SOURCE PROGRAM

This section describes the commands used to edit and assemble a Source Program. The commands are conversational -- OMEGA requests a command and some parameters with a question mark, and immediately either accepts or rejects the response.

Each command line on the Teletype is terminated with a Period. If OMEGA rejects the command, it will type out a Back-Arrow. Similarly, typing in a Back-Arrow indicates that the current command line should be abandoned without processing.

In the command descriptions, lowercase letters imply some number, and an underline indicates a type-out from OMEGA.

Two kinds of source statement lines are manipulated by commands: Input Lines, and Buffer Lines.

An Input Line Number is a decimal number between 1 and 32767. Leading zeroes are optional.

The Buffer is the memory above OMEGA used to build an edited Source Program. A command which refers to the Buffer can use a Buffer Line Number as low as 0 -- that is, just before the first line in the Buffer -- and as high as the current number of the final line. Because the Final Line Number is not always known exactly, the letter F can be used instead.

CONNECT DEVICE (C)

```
CId.
COd.
CLd.
```

The C command connects an OMEGA logical device to a physical device, or to the Buffer. You can make all the connections just once, after loading OMEGA, or you can change a connection whenever OMEGA asks for a new command.

Source Input Devices:

I1	Teletype Keyboard
I2	Teletype Paper Tape Reader
I3	HS Paper Tape Reader
I4	Card Reader
I5	Buffer (as Input for X command)
I6	Card Reader with Distributed I/O
I7	HS Paper Tape Reader with Distributed I/O
I0	Punch EOF Now

Punch Output Devices:

O1	Teletype Punch
O2	--
O3	HS Paper Tape Punch
O4	HS Paper Tape Punch with Distributed I/O
O0	(No Punch Output)

List Output Devices:

L1	Teletype Printer
L2	Data Products Printer
L3	Centronics Printer
L4	Data Products Printer with Distributed I/O
L5	Centronics Printer with Distributed I/O
L0	(No List Output)

You can enter several connections with one C command, by using one blank after each device:

```
CI4 O3 L2.

CI1 O0.
```

When OMEGA is first loaded, automatic connections are made to the Teletype, equivalent to this command:

```
CI2 O1 L1.
```

INITIALIZE (I)

I.

The I command initializes OMEGA for input and editing. The Buffer is cleared, and the last Input Line Number is set to 0. This command has no effect upon the Device Connections or the High Memory Limit.

An I command is automatically simulated when OMEGA is first loaded, and when an E command is entered.

RESTART

You can restart OMEGA at any time, and make it abandon any reading, printing, punching, or assembly in progress. No initialization is done for a restart; the Buffer, the Input Line Count, the Device Connections, and the High Memory Limit are intact.

There are three ways to cause a restart:

On an LSI-2, hit INT.
On an ALPHA-16, hit AUTO.
On all machines, hit STOP, set P to :0100, clear STOP, and hit RUN.

OMEGA will respond immediately with "?" and wait for the next command.

SET END OF MEMORY (E)

Ehhhh.

The E command resets OMEGA's High Memory Limit. When OMEGA is first loaded, it determines the size of memory, subtracts 16 words to allow for your bootstrap loader, and calls the result the end of available memory. If you want to protect more high memory than 16 words, enter a new hexadecimal address.

The E command triggers an automatic I command, clearing the Buffer and setting the last Input Line Number to 0.

If you need an E command every time you load OMEGA, you should probably create a new version of OMEGA with a fixed High Memory Limit. Refer to section on OMEGA Program Variables.

For LSI-3/05 with Software Console loaded, setting the end of memory below the Software Console will preserve the accessability of machine console.

READ INPUT (R)

```
Rm.
```

The R command reads thru Input Line m, and adds the lines to the Buffer. If Input Line m has already been passed, the command is rejected.

The last Input Line added is typed out for verification. If the end of the Source Input is found before Line m is reached, this message is also typed:

```
END OF TAPE: LINE NO mmmm
```

You can read in all of the Source Input by entering R9999. Alternatively, you can read the Source Input one piece at a time, with S or A commands between the R commands, as illustrated on the opposite page.

SKIP INPUT (S)

```
Sm n.
Sm.
```

The S command skips over Input Lines m thru n (inclusive), or -- for Sm. -- skips only Line m. If Input Line m has already been passed, the command is rejected.

If Line m is not the very next Input Line, all of the Source Input up to -- but not including -- Line m is read and added to the Buffer, as if an R command had been entered first.

The first and last Input Lines skipped are typed out for verification.

After an S command, you can replace the skipped lines immediately with an A command, or continue with more R and S commands, as illustrated on the opposite page.

```
         (INPUT)                        (TELETYPE)                    (BUFFER)

                                ?R5.
INPUT (LINE 001)                                              0001 INPUT (LINE 001)
INPUT (LINE 002)                                              0002 INPUT (LINE 002)
INPUT (LINE 003)                                              0003 INPUT (LINE 003)
INPUT (LINE 004)                                              0004 INPUT (LINE 004)
INPUT (LINE 005)                                              0005 INPUT (LINE 005)
                                     INPUT (LINE 005)

                                ?S9 13.
INPUT (LINE 006)                                              0006 INPUT (LINE 006)
INPUT (LINE 007)                                              0007 INPUT (LINE 007)
INPUT (LINE 008)                                              0008 INPUT (LINE 008)
                                     INPUT (LINE 009)
INPUT (LINE 009)
INPUT (LINE 010)
INPUT (LINE 011)
INPUT (LINE 012)
INPUT (LINE 013)
                                     INPUT (LINE 013)

                                ?R15.
INPUT (LINE 014)                                              0009 INPUT (LINE 014)
INPUT (LINE 015)                                              0010 INPUT (LINE 015)
                                     INPUT (LINE 015)

                                ?
```

ADD AFTER BUFFER LINE (A)

```
Am.
```

The A command opens the keyboard so you can insert Buffer Lines immediately after Buffer Line m. Type in successive lines of the Source Program, and end each line with a Carriage Return. To terminate the additions, enter a Carriage Return alone.

Backspace over typing errors with one or more Back-Arrows. Cancel a whole line by ending it with a Back-Arrow and a Carriage Return.

To insert lines before the first line currently in the Buffer, use A0. To add lines after the final line in the Buffer, use AF.

Remember that additions force re-numbering of all the Buffer Lines after the added lines, as illustrated on the opposite page. Add groups of lines from the bottom up.

DELETE BUFFER LINES (D)

```
Dm n.
Dm.
```

The D command deletes Buffer Lines m thru n (inclusive), or -- for Dm. -- deletes only Line m.

To delete the final line in the Buffer, use DF. To clear the entire Buffer, enter D1 F. The entire Buffer is also cleared when you enter the commands I, E, or B.

To replace a group of lines, first delete, then add:

```
?D41 42.
?A40.
REPLACEMENT FOR OLD 41  cr
REPLACEMENT FOR OLD 42  cr
cr
?
```

Remember that deletions force re-numbering of all the Buffer Lines after the deleted lines, as illustrated on the opposite page. Delete groups of lines from the bottom up.

```
         (BUFFER)                         (TELETYPE)

 0001    INPUT (LINE 001)
 0002    INPUT (LINE 002)
 0003    INPUT (LINE 003)
 0004    INPUT (LINE 004)
 0005    INPUT (LINE 005)
 0006    INPUT (LINE 006)
 0007    INPUT (LINE 007)
→0008    INPUT (LINE 008)
→0009    INPUT (LINE 014)
 0010    INPUT (LINE 015)
                             ?D8 9.
 0001    INPUT (LINE 001)
 0002    INPUT (LINE 002)
 0003    INPUT (LINE 003)
 0004    INPUT (LINE 004)
 0005    INPUT (LINE 005)
 0006    INPUT (LINE 006)
→0007    INPUT (LINE 007)
 0008    INPUT (LINE 015)
                             ?A6.
                             INSERTION 1 cr
                             INSERTION 2 cr
                             cr
                             ?
 0001    INPUT (LINE 001)
 0002    INPUT (LINE 002)
 0003    INPUT (LINE 003)
 0004    INPUT (LINE 004)
 0005    INPUT (LINE 005)
 0006    INPUT (LINE 006)
 0007    INSERTION 1
 0008    INSERTION 2
 0009    INPUT (LINE 007)
 0010    INPUT (LINE 015)
```

BUFFER CLEAR (B)

```
B.
```

The B command deletes all the lines in the Buffer. The commands I and E also clear the Buffer completely.

LIST BUFFER LINES (L)

```
Lm n.
Lm.
```

The L command lists Buffer Lines m thru n (inclusive), or -- for Lm. -- lists only Line m. To list the final line in the Buffer, use LF. To list the entire Buffer, enter L1 F.

Each L command produces a new formatted listing. Each Buffer Line is preceded by its current Line Number. Each page has 54 printed lines and 11 blank lines. If Device L is connected to a Teletype, the final (or only) page is not formatted. This saves paper if the listing is less than one page long.

PUNCH BUFFER LINES (P)

```
P m n.
P m.
```

The P command punches Buffer Lines m thru n (inclusive), or -- for P m. -- punches Line m only. Note the blank required before the specification for m.

Each line punched is terminated with the sequence:

Carriage Return
Line Feed
Null

This sequence makes manual splicing easier, and is also suitable for re-entering the tape to OMEGA.

To punch the final line in the Buffer, use P F.

Some blank leader will be included if you follow the letter P with an L.

An Up-Arrow and some blank trailer will be included if you follow the P (or the L) with a T. The Up-Arrow represents End-of-Tape to OMEGA. If the tape is later re-entered with an R command, reading will stop at the Up-Arrow. If the tape is fed directly into an X command, the Up-Arrow will allow another piece of input to be used, unless an END statement was on the current piece.

To punch a complete program from the Buffer for future use, enter:

```
?PLT 1 F.
```

RESET LAST INPUT LINE NUMBER (T)

Tm.

The T command is used to re-synchronize the Input Line Numbers with an assembly listing, or with your latest listing of the Buffer Lines. This is quite useful when you're building a new program from several pieces of tape, or when a series of R and S commands has allowed the Source Input to get out of synch with the Buffer.

OMEGA uses the Line Number in the command as the number of the last Input Line already passed. The next line about to be accessed by an R or S command is therefore m + 1.

The value of m can be 0, making the next Input Line into Line Number 1. This setting is made automatically for any of these conditions:

OMEGA just loaded.
I command entered.
E command entered.

MEMORY AVAILABLE DISPLAY (M)

```
        M.
```

The M command simply asks OMEGA to type out the amount of memory still available between the fixed part of the assembler and the High Memory Limit. The number displayed is the decimal count of the words (not bytes) left for building Buffer Lines, Symbol Table entries, and Literal Pools.

If a source statement is added to the Buffer with an R or an S command, every two characters consume one word of memory. A Carriage Return is appended to each line, but extra blanks between the fields are compressed out.

A statement added to the Buffer with an A command is not compressed, and should be typed in with only one blank separating adjacent fields. Similarly, a source statement fed directly to an X command thru the Teletype keyboard is not compressed as it is when read from other devices.

After an X command is entered, and assembly begins, each new Label, SET Variable, New Op Code Definition, or Literal needs 4 words of memory.

If the memory available is exhausted during Buffer editing, this message appears on the Teletype:

```
        BUFFER FULL: LAST SOURCE LINE IS mmmm
```

An M command would show a very low number of words left. Either delete a substantial number of characters from the Buffer (perhaps a page of Comment Lines, or a piece of the Source Program not currently needed); or punch out a partial Source Program, clear the Buffer, build the rest of the program, punch it out, initialize OMEGA, and assemble from the complete tape:

```
        ?PL 1 F.

        ?B.

        ?    (T, R, S, A, and D commands)

        ?PT 1 F.

        ?I.

        ?CL3 I3.

        ?X.
```

EXECUTE ASSEMBLER (X)

```
X.
XE.
XL.
X2.
```

The X command ends the interactive editing of a Source Program, and begins an actual assembly. No more commands are accepted until an END statement has been processed.

If the Buffer is not empty, you can connect it to Device I and assemble your edited program directly:

```
?CI5 L3 O3.
?X.
```

To protect you against destroying an edited Buffer, OMEGA will not accept an X command if Device I is connected to anything except the Buffer, as long as the Buffer has some lines in it. For an assembly from cards or paper tape after an editing session, initialize OMEGA, connect the reader, and start the assembly:

```
?I.
?CI3 L3 O3.
?X.
```

A normal assembly, requested with a simple X command, does three things:

1. Performs two passes over the Source Input.
2. Generates a complete listing.
3. Punches one Object Program followed by an EOF.

You can suppress all printed output by connecting Device L to 0, or all punched output by connecting Device O to 0:

```
?CL0.
?CO0.
```

You can restrict the listing to only Error Lines by inserting the letter E before the period in the command:

```
?XE.
```

If each new Object Program you're punching is part of an Object Program Library tape, you don't want an EOF following each program. Specify Library Format for the Object Program by inserting the letter L before the period:

```
?XL.
```

OMEGA will immediately punch one EOF on its output tape whenever you enter this special command:

```
?CI0.
```

This lets you use a consistent XL command for a series of assemblies, and explicitly supply the punched EOF later.

Once an assembly has terminated, you can produce another copy of the printed and punched output by requesting OMEGA to repeat Pass 2 only:

```
    ?X2.
```

Connections may be changed before each X2 command. For example, you may want another listing, but not another punched Object Program:

```
    ?CI5 L3 O3.
    ?XL.
    ?CO0.
    ?X2.
```

The modifiers E, L, and 2 can be combined in any order after the X and before the period:

```
    XEL.
    X2E.
    XLE2.
```

OMEGA PROGRAM VARIABLES

Certain fixed locations in low memory contain values which control the operation of OMEGA. Each value may be changed immediately after loading OMEGA, and a new paper tape which preserves the modifications may be punched with BDP, the Binary Dump Program.

High Memory Limit

When OMEGA is first loaded and executed, the high end of memory is determined, :0010 is subtracted, and the result is stored at location :0002 Absolute. Unless an E command is used to change the value later, OMEGA will use the stored address as the upper limit of its available memory.

To prevent OMEGA from making the initial calculation, replace the JST at location :0100 Absolute with a NOP. Set location :0002 Absolute to the new fixed High Memory Limit.

MACH Value

If no MACH statement is supplied to OMEGA2, it uses the initial contents of location :0003 Absolute as the MACH value. The distributed version of OMEGA2 has :0002 -- binary 010 -- at this location, indicating the LSI-1 instruction set.

Lines per Page

The maximum number of lines in the body of a page is carried as a negative number in location :0004 Absolute. The distributed version of OMEGA uses :FFCB, or -53. This value allows 13 lines for the top and bottom margins, and for the page heading and title.

Characters per Line

The maximum number of characters on each line of the assembly listing is carried as a negative number in location :0005 Absolute. The distributed version of OMEGA uses :FFB8, or -72.

OMEGA COMMAND SUMMARY

CONTROL

I.	Initialize OMEGA -- clear Buffer and reset last Input Line read to 0.
B.	Buffer clear.
Ehhhh.	End of memory set to hexadecimal address.
M.	Memory available displayed in decimal words.
X.	Execute assembler.
XE.	Error list only.
XL.	Library Format for Object Program -- no EOF.
X2.	Pass 2 again.

BUFFER EDITING

Am.	Add after Line m.
Dm.	Delete Line m.
Dm n.	Delete Lines m thru n.
Lm.	List Line m.
Lm n.	List Lines m thru n.
P m.	Punch Line m.
P m n.	Punch Lines m thru n.
PL m n.	With leader.
PT m n.	With trailer.
PLT m n.	With leader and trailer.

INPUT EDITING

Rm.	Read thru Line m, and add to end of Buffer.
Sm.	Skip Line m, after reading thru Line m-1.
Sm n.	Skip Lines m thru n, after reading thru Line m-1.
Tm.	Reset last Input Line read to m.

LOGICAL DEVICES

Cd. Connect devices:

Source Input		Punch Output		List Output	
I1	Teletype Keyboard	O1	Teletype	L1	Teletype
I2	Teletype Paper Tape	O2	N/A	L2	Data Products
I3	HS Paper Tape	O3	HS Paper Tape	L3	Centronics
I4	Card Reader	O4	HS Paper Tape (DIO)	L4	Data Products (DIO)
I5	Buffer Lines to X.	O0	No Punching	L5	Centronics (DIO)
I6	Card Reader (DIO)			L0	No Listing
I7	HS Paper Tape (DIO)				
I0	EOF Now				

Section 13

MESSAGES ON THE TELETYPE

<u>OMEGAn (rr)</u>

CAUSE: OMEGA has begun execution. Revision level of the program is rr.
ACTION: None.

<u>FEED ME: RUN</u>

CAUSE: The assembler could not save the source statements read during Pass 1, because the Symbol Table needed the memory.
ACTION: Reposition the Source Program tape to the start of the last program read, and hit RUN.

<u>PAUSE</u>

CAUSE: Input ended with an up-arrow, indicating that more is to follow.
ACTION: Ready the next piece of input, and hit RUN.

<u>PUNCH ON, RUN. AT HALT OFF, RUN.</u>

CAUSE: The Teletype punch is about to be used.
ACTION: Turn on the punch and hit RUN. At the next machine halt, turn off the punch and hit RUN again.

<u>RECORD GT 80 CHARACTERS</u>

CAUSE: An assembler language source statement was expected, but the tape record was too long. The unacceptable tape is probably either an improperly delimited header, or an Object Program.
ACTION: Correct the problem, and enter appropriate commands to continue editing or assembly.

<u>NO 'END' DIRECTIVE</u>

CAUSE: Input to an X command has reached EOF before an END statement was processed.
ACTION: Edit the input into acceptable format, and repeat the assembly.

Appendix A

ASCII Character Set

Graphic	Hex Value	Card Code
Blank	:A0	Blank
!	:A1	11-2-8
"	:A2	7-8
#	:A3	3-8
$	:A4	11-3-8
%	:A5	0-4-8
&	:A6	12
'	:A7	5-8
(	:A8	12-5-8
)	:A9	11-5-8
*	:AA	11-4-8
+	:AB	12-6-8
,	:AC	0-3-8
-	:AD	11
.	:AE	12-3-8
/	:AF	0-1
0	:B0	0
1	:B1	1
2	:B2	2
3	:B3	3
4	:B4	4
5	:B5	5
6	:B6	6
7	:B7	7
8	:B8	8
9	:B9	9
:	:BA	2-8
;	:BB	11-6-8
<	:BC	12-4-8
=	:BD	6-8
>	:BE	0-6-8
?	:BF	0-7-8
@	:C0	4-8

Graphic	Hex Value	Card Code
A	:C1	12-1
B	:C2	12-2
C	:C3	12-3
D	:C4	12-4
E	:C5	12-5
F	:C6	12-6
G	:C7	12-7
H	:C8	12-8
I	:C9	12-9
J	:CA	11-1
K	:CB	11-2
L	:CC	11-3
M	:CD	11-4
N	:CE	11-5
O	:CF	11-6
P	:D0	11-7
Q	:D1	11-8
R	:D2	11-9
S	:D3	0-2
T	:D4	0-3
U	:D5	0-4
V	:D6	0-5
W	:D7	0-6
X	:D8	0-7
Y	:D9	0-8
Z	:DA	0-9
[	:DB	0-2-8
\	:DC	11-7-8
]	:DD	0-5-8
↑	:DE	12-2-8
←	:DF	12-7-8

Appendix B

MACHINE INSTRUCTION SETS

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
AAI	2		X	X	X
ADD	1	X	X	X	X
ADDB	8	X	X	X	X
ADDS	10			X	
AIB	6	X	X	X	X
AIN	6	X	X	X	X
ALA	4	X	X	X	
ALX	4	X	X	X	
ANA	5	X	X	X	
AND	1	X	X	X	X
ANDB	8	X	X	X	X
ANDS	10			X	
ANX	5	X	X	X	
AOB	6	X	X	X	X
AOT	6	X	X	X	X
ARA	4	X	X	X	
ARM	5	X	X	X	
ARP	5	X	X	X	
ARX	4	X	X	X	
AXI	2	X	X	X	X
AXM	5	X	X	X	
AXP	5	X	X	X	
BAO	4		X	X	
BCA	5			X	
BCX	5			X	
BIN	6	X	X	X	
BOT	6	X	X	X	
BSA	5			X	
BSX	5			X	
BXO	4		X	X	
CAI	2	X	X	X	X
CAR	5	X	X	X	
CAX	5	X	X	X	
CID	5	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
CIE	5	X	X	X	X
CMS	1	~~X~~	X	X	X
CMSB	8	X	X	X	X
CMSS	10			X	
COV	5	X	X	X	
CXA	5	X	X	X	
CXI	2	X	X	X	X
CXR	5	X	X	X	
DAR	5	X	X	X	
DAX	5	X	X	X	
DIN	5	X	X	X	X
DVD	9		X	X	
DVS	7	X			
DXA	5	X	X	X	
DXR	5	X	X	X	
EAX	5		X	X	
EIN	5	X	X	X	X
EIX	5			X	
EMA	1	X	X	X	X
EMAB	8	X	X	X	X
EMAS	10			X	
HLT	5	X	X	X	X
HTR	5				X
IAR	5	X	X	X	
IAX	5	X	X	X	
IBA	6	X	X	X	
IBAM	6	X	X	X	
IBX	6	X	X	X	
IBXM	6	X	X	X	
ICA	5		X	X	X
ICX	5		X	X	X
IMS	1	X	X	X	X
IMSS	10			X	
INA	6	X	X	X	X
INAM	6	X	X	X	
INX	6	X	X	X	X
INXM	6	X	X	X	
IOR	1	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI 3/05
IORB	8	X	X	X	X
IORS	10	--		X	
IPX	5		X	X	
ISA	6	X	X	X	X
ISX	6	X	X	X	X
IXA	5	X	X	X	
IXR	5	X	X	X	
JAG	3	X	X	X	X
JAL	3	X	X	X	X
JAM	3	X	X	X	X
JAN	3	X	X	X	X
JAP	3	X	X	X	X
JAZ	3	X	X	X	X
JMP	3	X	X	X	X
JMPS	10			X	
JOC	3	X	X	X	
JOR	3	X	X	X	X
JOS	3	X	X	X	X
JSR	3	X	X	X	X
JSS	3	X	X	X	X
JST	3	X	X	X	X
JSTS	10			X	
JXN	3	X	X	X	X
JXZ	3	X	X	X	X
LAM	2	X	X	X	X
LAO	5	X	X	X	
LAP	2	X	X	X	X
LDA	1	X	X	X	X
LDAB	8	X	X	X	X
LDAS	10			X	
LDX	1	X	X	X	X
LDXB	8	X	X	X	X
LDXS	10			X	
LLA	4	X	X	X	X
LLL	7	X	X	X	
LLR	7	X	X	X	
LLX	4	X	X	X	X
LRA	4	X	X	X	X
LRL	7	X	X	X	
LRR	7	X	X	X	
LRX	4	X	X	X	X
LXM	2	X	X	X	X
LXO	5	X	X	X	
LXP	2	X	X	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
MPS	7	X			
MPY	9	——	X	X	
NAR	5	X	X	X	X
NAX	5	X	X	X	X
NOP	5	X	X	X	X
NOR	4	X			
NRM	9		X	X	
NRA	5	X	X	X	
NRX	5	X	X	X	
NXA	5	X	X	X	X
NXR	5	X	X	X	X
OCA	6		X	X	X
OCX	6		X	X	X
OTA	6	X	X	X	X
OTX	6	X	X	X	X
OTZ	6	X	X	X	
PFD	5	X	X	X	
PFE	5	X	X	X	
RBA	6	X	X	X	
RBAM	6	X	X	X	
RBX	6	X	X	X	
RBXM	6	X	X	X	
RDA	6	X	X	X	
RDAM	6	X	X	X	
RDX	6	X	X	X	
RDXM	6	X	X	X	
RLA	4	X	X	X	X
RLX	4	X	X	X	X
ROV	5	X	X	X	X
RRA	4	X	X	X	X
RRX	4	X	X	X	X
RTCD	5				X
RTCE	5				X
SAI	2		X	X	X
SAO	5	X	X	X	
SBM	5	X	X	X	
SCM	1	X	X	X	
SCMB	8	X	X	X	
SCN	1	X			
SEA	6	X	X.	X	X

Assembler Mnemonic	Syntax Class	Alpha 16	LSI-1	LSI-2 /10, /20	LSI-3/05
SEL	6	X	X	X	
SEN	6	X	X	X	X
SEX	6	X	X	X	X
SIA	5	X	X	X	X
SIN	4	X	X	X	X
SIX	5	X	X	X	X
SLAS	10			X	
SOA	5	X	X	X	X
SOV	5	X	X	X	X
SOX	5	X	X	X	X
SSN	5	X	X	X	
STA	1	X	X	X	X
STAB	8	X	X	X	X
STAS	10			X	
STOP	2	X	X	X	X
STX	1	X	X	X	X
STXB	8	X	X	X	X
STXS	10			X	
SUB	1	X	X	X	X
SUBB	8	X	X	X	X
SUBS	10			X	
SWM	5	X	X	X	X
SXI	2	X	X	X	X
SXO	5	X	X	X	
TAX	5	X	X	X	X
TPX	5				X
TRP	5	X	X	X	
TXA	5	X	X	X	X
WAIT	5	X	X	X	
WRA	6	X	X	X	
WRX	6	X	X	X	
WRZ	6	X	X	X	
XOR	1	X	X	X	X
XORB	8	X	X	X	X
XORS	10			X	
XRM	5	X	X	X	
XRP	5	X	X	X	
ZAR	5	X	X	X	
ZAX	5	X	X	X	
ZXR	5	X	X	X	

Appendix C

~~LSI~~-2 INSTRUCTIONS

This appendix contains the machine code layouts for all the instructions available on the LSI-1, LSI-2/10, and LSI-2/20.

The instructions are grouped by standard assembler Syntax Class, and the Mnemonics are alphabetized within each class. For the programmer's convenience, the syntax charts from Section 3 are reproduced.

Class	Machine Function
1	Word Reference
2	Byte Immediate
3	Conditional Jump
4	Single Register Bit Change
5	Register and Control
6	Input/Output
7	Double Register Bit Change
8	Byte Reference
9	Double Register Arithmetic
10	Stack Reference

For a detailed description of each instruction function, the programmer should refer to the CA publication entitled Computer Handbook.

CLASS 1: WORD REFERENCE

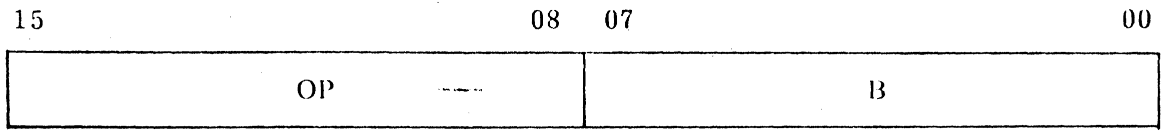
15 — 11	10 — 09	08	07 — 00
OP	M	I	D

OP	Operation Code
M	Addressing Mode: 00 Scratchpad: D 01 Relative Forward: P+D 10 Indexed: X+D 11 Relative Backward: P-1-D
I	Indirect Address Flag
D	Displacement

MNEMONIC [* / @ / *@ / =] OPERAND

Skeleton	Mnemonic	Function
8800	ADD	Add to A
8000	AND	AND to A
D000	CMS	Compare A with Memory, Skip (Low, High, Equal)
B800	EMA	Exchange Memory with A
D800	IMS	Increment Memory, Skip on Zero
A000	IOR	Inclusive OR to A
F000	JMP	Jump Unconditional
F800	JST	Jump and Store P
B000	LDA	Load A
E000	LDX	Load X
CD00	SCM	Scan Memory
9800	STA	Store A
E800	STX	Store X
9000	SUB	Subtract from A
A800	XOR	Exclusive OR to A

CLASS 2: BYTE IMMEDIATE



OP	Operation Code
B	Byte Immediate Value

MNEMONIC | OPERAND

Hex	Mnemonic	Function
0B00	AAI	Add to A Immediate
C200	AXI	Add to X Immediate
C000	CAI	Compare to A Immediate, Skip on Not Equal
C100	CXI	Compare to X Immediate, Skip on Not Equal
C700	LAM	Load A Minus Immediate
C600	LAP	Load A Positive Immediate
C500	LXM	Load X Minus Immediate
C400	LXP	Load X Positive Immediate
0D00	SAI	Subtract from A Immediate
C300	SXI	Subtract from X Immediate
0800	STOP	Stop

CLASS 3: CONDITIONAL JUMP

15 – 13	12	11 – 07	06	05 – 00
OP	G	C	FB	D

OP Operation Code

G Group Test:

0 OR
1 AND

C

Condition Bit		G = 0	G = 1
11	Magnitude of X	X = 0	X ≠ 0
10	SENSE	Reset	Set
09	OV	Set (Resets OV)	Reset
08	Magnitude of A	A = 0	A ≠ 0
07	Sign of A	A Negative	A Positive

FB Jump Direction:

0 Forward
1 Backward

D Jump Distance:

Forward	P+D
Backward	P-1-D

MNEMONIC	OPERAND

SPECIAL CASE

JOC	GC,OPERAND

GC is an absolute expression which specifies all the bits of the G and C fields.

Skeleton	Mnemonic	Function: Jump When
3180	JAG	A Greater than Zero
2180	JAL	A Less than, or Equal to, Zero
2080	JAM	A Minus
3100	JAN	~~A~~ Not Zero
3080	JAP	A Positive
2100	JAZ	A Zero
3200	JOR	OV Reset
2200	JOS	OV Set (and Force OV Reset)
2400	JSR	SENSE Reset
3400	JSR	SENSE Set
3800	JXN	X Not Zero
2800	JXZ	X Zero
2000	JOC	Conditions

CLASS 4: SINGLE REGISTER BIT CHANGE

15 — 03	02 — 00
OP	C

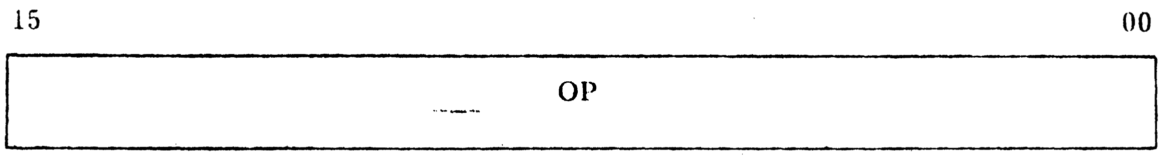
OP Operation Code

C For most instructions, C = Operand-1
For SIN N, C = N+1
For BAO N and for BXO N:
If N is 0 thru 7, C = N
If N is 8 thru 15, C = 15-N

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
1050	ALA	Arithmetic Left A
1028	ALX	Arithmetic Left X
10D0	ARA	Arithmetic Right A
10A8	ARX	Arithmetic Right X
1340	BAO	Bit of A to OV (15 thru 8)
13C0	BAO	Bit of A to OV (0 thru 7)
1320	BXO	Bit of X to OV (15 thru 8)
13A0	BXO	Bit of X to OV (0 thru 7)
1350	LLA	Logical Left A
1328	LLX	Logical Left X
13D0	LRA	Logical Right A
13A8	LRX	Logical Right X
1150	RLA	Rotate Left A
1128	RLX	Rotate Left X
11D0	RRA	Rotate Right A
11A8	RRX	Rotate Right X
6800	SIN	Status Inhibit

CLASS 5: REGISTER AND CONTROL



OP Operation Code

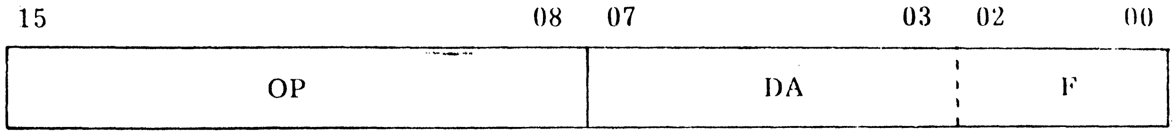
MNEMONIC [COMMENTS]

Skeleton	Mnemonic	Function
0070	ANA	AND of A and X to A
0068	ANX	AND of A and X to X
0010	ARM	Set A to -1
0350	ARP	Set A to +1
0018	AXM	Set A and X to -1
0358	AXP	Set A and X to +1
06CA	BCA	Bit Clear A
06C8	BCX	Bit Clear X
068A	BSA	Bit Set A
0688	BSX	Bit Set X
9210	CAR	Complement A
0208	CAX	Complement A and put in X
1600	COV	Complement OV
0410	CXA	Complement X and put in A
0408	CXR	Complement X
00D0	DAR	Decrement A
00C8	DAX	Decrement A and put in X
00B0	DXA	Decrement X and put in A
00A8	DXR	Decrement X
0428	EAX	Exchange A with X
0218	EIX	Execute Instruction pointed to be X
0510	IAR	Increment A
0148	IAX	Increment A and put in X
5804	ICA	Input Console Data Register to A
5A04	ICX	Input Console Data Register to X
0090	IPX	Increment P and put in X
5801	ISA	Input Console Sense Register to A
5A01	ISX	Input Console Sense Register to X
0130	IXA	Increment X and put in A
0128	IXR	Increment X
13C0	LAO	Least significant bit of A to OV
13A0	LXO	Least significant bit of X to OV
0310	NAR	Negate A
0308	NAX	Negate A and put in X

Skeleton	Mnemonic	Function
0610	NRA	NOR of A and X to A
0608	NRX	NOR of A and X to X
1510	NXA	Negate X and put in A
0508	NXR	Negate X
4404	OCA	Output A to Console Data Register
4406	OCX	Output X to Console Data Register
1200	ROV	Reset OV
1340	SAO	Sign of A to OV
1400	SOV	Set OV
1320	SXO	Sign of X to OV
0048	TAX	Transfer A to X
0030	TXA	Transfer X to A
0008	XRM	Set X to -1
0528	XRP	Set X to +1
0110	ZAR	Zero A
0118	ZAX	Zero A and X
0108	ZXR	Zero X

4006	CID	Console Interrupt Disable
4005	CIE	Console Interrupt Enable
0C00	DIN	Disable Interrupts
0A00	EIN	Enable Interrupts
0800	HLT	Halt
0000	NOP	No Operation
4003	PFD	Power Fail Interrupt Disable
4002	PFE	Power Fail Interrupt Enable
0E00	SBM	Set Byte Mode
5800	SIA	Status Input to A
5A00	SIX	Status Input to X
6C00	SOA	Status Output from A
6E00	SOX	Status Output from X
0F00	SWM	Set Word Mode
4007	TRP	Trap
F600	WAIT	Wait for Interrupts

CLASS 6: INPUT/OUTPUT



OP	Operation Code
DA	Device Address
F	Function Code

(This is the nominal division of bits 07 -- 00. The exact interpretation of the bits is left to the device logic.)

MNEMONIC OPERAND[,OPERAND]

Skeleton	Mnemonic	Function
5400	AIB	Automatic Input to Memory -- Byte
5000	AIN	Automatic Input to Memory -- Word
6400	AOB	Automatic Output from Memory -- Byte
6000	AOT	Automatic Output from Memory -- Word
7100	BIN	Block Input to Memory
7500	BOT	Block Output from Memory
7800	IBA	Input Byte to A
7C00	IBAM	Input Byte to A Masked
7A00	IBX	Input Byte to X
7E00	IBXM	Input Byte to X Masked
5800	INA	Input Word to A
5C00	INAM	Input Word to A Masked
5A00	INX	Input Word to X
5E00	INXM	Input Word to X Masked
6C00	OTA	Output A
6E00	OTX	Output X
6800	OTZ	Output Zeros
7900	RBA	Read Byte to A
7D00	RBAM	Read Byte to A Masked
7B00	RBX	Read Byte to X
7F00	RBXM	Read Byte to X Masked
5900	RDA	Read Word to A
5D00	RDAM	Read Word to A Masked
5B00	RDX	Read Word to X
5F00	RDXM	Read Word to X Masked

Skeleton	Mnemonic	Function
4400	SEA	Select and Present A
4000	SEL	Select
4900	SEN	Sense and Skip on Response
4600	SEX	Select and Present X
4800	SSN	Sense and Skip on No Response
6D00	WRA	Write from A
6F00	WRX	Write from X
6900	WRZ	Write Zeros

CLASS 7: DOUBLE REGISTER BIT CHANGE

15 04	03 00
OP	C

OP	Operation Code
C	Operand-1

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
1B00	LLL	Long Logical Left
1B80	LLR	Long Logical Right
1900	LRL	Long Rotate Left
1980	LRR	Long Rotate Right

CLASS 8: BYTE REFERENCE

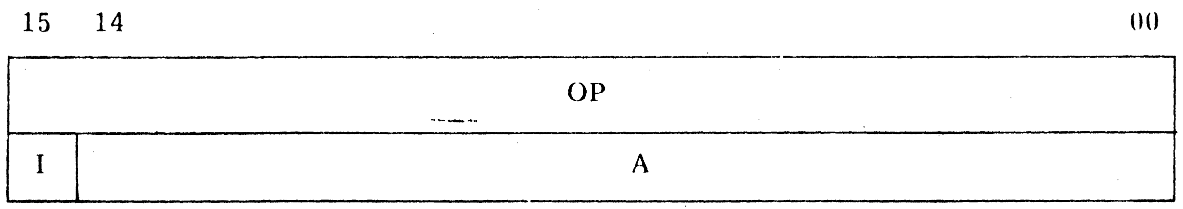
15 — 11	10 — 08	07 — 00
OP	M/I	D

OP	Operation Code
M/I	Addressing Mode and Indirect Address Flag:
	000 Scratchpad Byte: D
	010 Relative Forward, Byte 0 of Word: P+D
	100 Indexed Byte: X+D
	110 Relative Forward, Byte 1 of Word: P+D
	001 Indirect Scratchpad: *D
	011 Indirect Relative Forward: *(P+D)
	101 Indirect Scratchpad Post-Indexed: *D+X
	111 Indirect Relative Backward: *(P-1-D)
D	Displacement

MNEMONIC $\begin{bmatrix} * \\ @ \\ *@ \end{bmatrix}$ OPERAND

Skeleton	Mnemonic	Function
8800	ADDB	Add to A
8000	ANDB	AND to A
D000	CMSB	Compare A with Memory, Skip (Low, High, Equal)
B800	EMAB	Exchange Memory with A
A000	IORB	Inclusive OR to A
B000	LDAB	Load A
E000	LDXB	Load X
CD00	SCMB	Scan Memory
9800	STAB	Store A
E800	STXB	Store X
9000	SUBB	Subtract from A
A800	XORB	Exclusive OR to A

CLASS 9: DOUBLE REGISTER ARITHMETIC

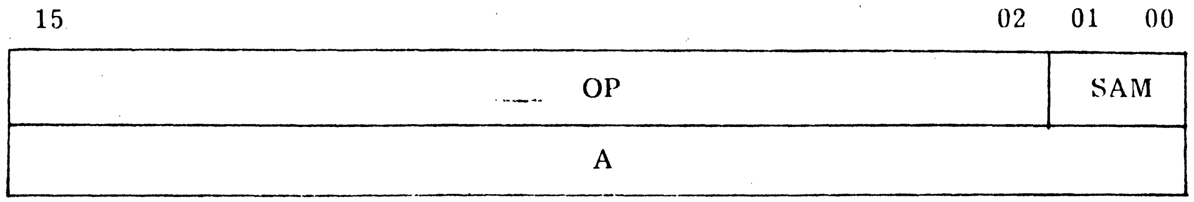


OP	Operation Code
I	Indirect Address Flag
A	Address of Operand

MNEMONIC [*]OPERAND

Skeleton	Mnemonic	Function
1970	DVD	Divide
1960	MPY	Multiply and Add
1940	NRM	Normalize

CLASS 10: STACK REFERENCE



OP	Operation Code
A	Address of Operand
SAM	Stack Address Mode:

Value	Symbol	Mode
00	blank	Direct (Value of Pointer)
01	,@	Indexed (Pointer + X)
10	,+	Pop (Increment Pointer After Access)
11	,-	Push (Decrement Pointer Before Access)

MNEMONIC OPERAND $\left[\begin{array}{l},@\\,+\\,-\end{array}\right]$

Skeleton	Mnemonic	Function ("SE" means "Stack Element")
1438	ADDS	Add SE to A
1418	ANDS	AND SE to A
1658	CMSS	Compare A with SE, Skip (Low, High, Equal)
14F8	EMAS	Exchange A with SE
1678	IMSS	Increment SE, Skip on Zero
1498	IORS	Inclusive OR SE to A
16D8	JMPS	Jump Unconditional to SE
16F8	JSTS	Jump and Store P to SE
14D8	LDAS	Load A from SE
16B8	LDXS	Load X from SE
1618	SLAS	SE Location to A
1478	STAS	Store A into SE
16B8	STXS	Store X into SE
1458	SUBS	Subtract SE from A
14B8	XORS	Exclusive OR SE to A

Appendix D

LSI-3/05 INSTRUCTIONS

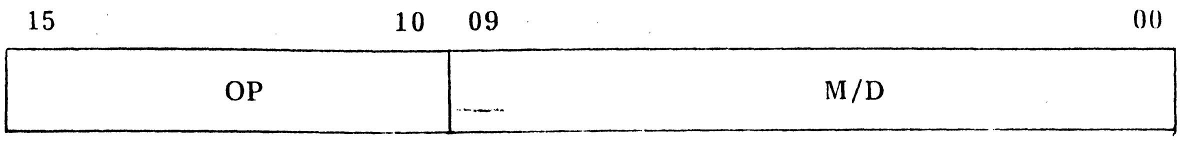
This appendix contains the machine code layouts for all the instructions available on the LSI-3/05.

The instructions are grouped by standard assembler Syntax Class, and the Mnemonics are alphabetized within each class.

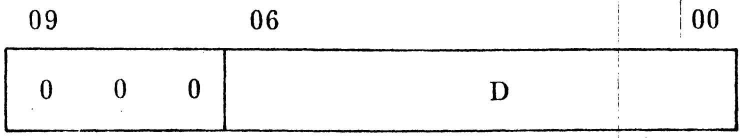
Class	Machine Function
1	Word Reference
2	Byte Immediate
3	Conditional Jump
4	Single Register Bit Change
5	Register and Control
6	Input/Output
8	Byte Reference

For a detailed description of each instruction function, the programmer should refer to the CA publication entitled Computer Handbook.

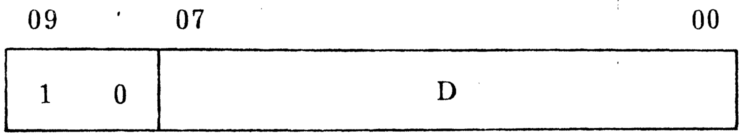
CLASS 1: WORD REFERENCE



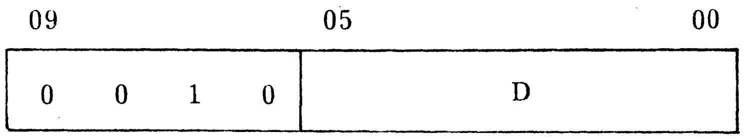
OP	Operation Code
M/D	Addressing Mode and Displacement



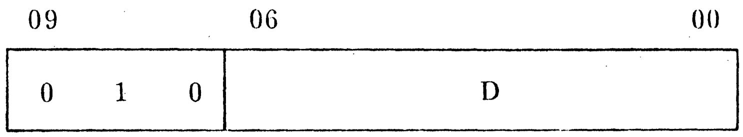
Scratchpad: D



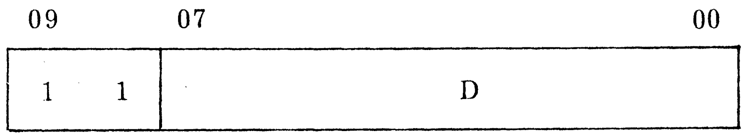
Relative: P+D-128



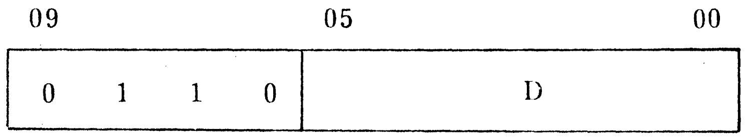
Indexed: X+D



Indirect Scratchpad: *D



Indirect Relative: *(P+D-128)



Indirect Scratchpad
Post-Indexed: *D+X

$$\text{MNEMONIC} \quad \begin{bmatrix} * \\ @ \\ *@ \\ = \end{bmatrix} \text{Operand}$$

Prefixes:

*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed
=	Literal Pool Reference

Skeleton	Mnemonic	Function
8800	ADD	Add to A
9400	AND	AND to A
B800	CMS	Compare A with Memory, Skip (Low, High, Equal)
9000	EMA	Exchange Memory with A
DC00	IMS	Increment Memory, Skip on Zero
B400	IOR	Inclusive OR to A
9C00	JMP	Jump Unconditional
BC00	JST	Jump and Store P
8000	LDA	Load A
A000	LDX	Load X
8400	STA	Store A
A400	STX	Store X
8C00	SUB	Subtract from A
9800	XOR	Exclusive OR to A

CLASS 2: BYTE IMMEDIATE

15 09	08	07 00
OP	F	B

OP	Operation Code
F	Flag for Operand Value F = 1 for: AAI/AXI LAP/LXP F = 0 for: CAI/CXI F = 1 when Operand = 0, but F = 0 otherwise, for: LAM/LXM SAI/SXI
B	Byte Immediate Value If F = 1, B = Operand If F = 0, B = 256-Operand

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
0B00	AAI	Add to A Immediate
2B00	AXI	Add to X Immediate
0C00	CAI	Compare to A Immediate, Skip on Not Equal
2C00	CXI	Compare to X Immediate, Skip on Not Equal
0800	LAM	Load A Minus Immediate
0900	LAP	Load A Positive Immediate
2800	LXM	Load X Minus Immediate
2900	LXP	Load X Positive Immediate
0A00	SAI	Subtract from A Immediate
2A00	SXI	Subtract from X Immediate
3C00	STOP	Stop

CLASS 3: CONDITIONAL JUMP

15 07	06 00
OP	D

OP Operation Code

D Destination: P-64+D

MNEMONIC OPERAND

Skeleton	Mnemonic	Function: Jump When
1200	JAG	A Greater than Zero
1280	JAL	A Less than, or Equal to, Zero
1380	JAM	A Minus
1180	JAN	A Not Zero
1300	JAP	A Positive
1100	JAZ	A Zero
3680	JOR	OV Reset
3600	JOS	OV Set (and Force OV Reset)
1680	JSR	SENSE Reset
1600	JSS	SENSE Set
3180	JXN	X Not Zero
3100	JXZ	X Zero

CLASS 4: SINGLE REGISTER BIT CHANGE

15 08	07 04	03 00
OP1	C	OP2

OP1	Operation Code, Part 1
C	Operand-1
OP2	Operation Code, Part 2

MNEMONIC OPERAND

Skeleton	Mnemonic	Function
0E01	LLA	Logical Left A
2E01	LLX	Logical Left X
0E09	LRA	Logical Right A
2E09	LRX	Logical Right X
0E03	RLA	Rotate Left A
2E03	RLX	Rotate Left X
0E0B	RRA	Rotate Right A
2E0B	RRX	Rotate Right X
0E0F	SIN	Status Inhibit

CLASS 5: REGISTER AND CONTROL

15	00
OP	

OP   Operation Code

MNEMONIC   [COMMENTS]

Skeleton	Mnemonic	Function
0104	ICA	Input Console Data Register to A
2104	ICX	Input Console Data Register to X
0101	ISA	Input Console Sense Register to A
2101	ISX	Input Console Sense Register to X
0001	NAR	Negate A
2001	NAX	Negate A and Put in X
0021	NXA	Negate X and Put in A
2021	NXR	Negate X
0404	OCA	Output A to Console Data Register
2404	OCX	Output X to Console Data Register
0E17	ROV	Reset OV
0E15	SOV	Set OV
2000	TAX	Transfer A to X
2010	TPX	Transfer P to X
0020	TXA	Transfer X to A
0E47	CID	Console Interrupt Disable
0E45	CIE	Console Interrupt Enable
0E87	DIN	Disable Interrupts
0E85	EIN	Enable Interrupts
0E0D	HLT	Halt
0080	HTR	Halt and Reset
0000	NOP	No Operation
0E57	RTCD	Real Time Clock Disable
0E55	RTCE	Real Time Clock Enable
0E25	SBM	Set Byte Mode
0030	SIA	Status Input to A
2030	SIX	Status Input to X
3000	SOA	Status Output from A
3020	SOX	Status Output from X
0E27	SWM	Set Word Mode

CLASS 6: INPUT/OUTPUT

15 — 08	07 — 03	02 — 00
OP	DA	F

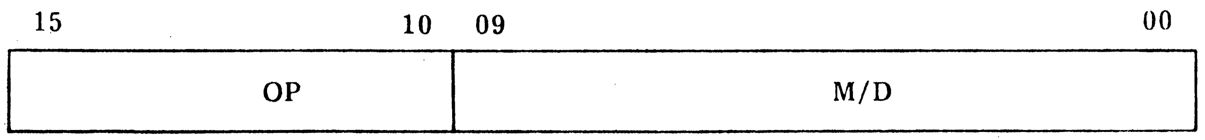
OP	Operation Code
DA	Device Address
FC	Function Code

(This is the nominal division of bits 07 -- 00. The exact interpretation of the bits is left to the device logic.)

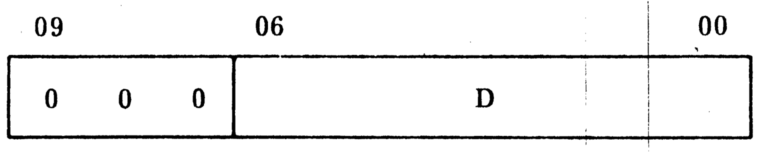
MNEMONIC OPERAND[,OPERAND]

Skeleton	Mnemonic	Function
4500	AIB	Automatic Input to Memory -- Byte
0500	AIN	Automatic Input to Memory -- Word
6500	AOB	Automatic Output from Memory -- Byte
2500	AOT	Automatic Output from Memory -- Word
0100	INA	Input Word to A
2100	INX	Input Word to X
0200	OTA	Output A
2200	OTX	Output X
0400	SEA	Select and Present A
0600	SEN	Sense and Skip on Response
2400	SEX	Select and Present X

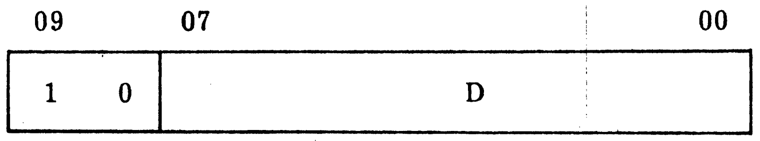
CLASS 8: BYTE REFERENCE



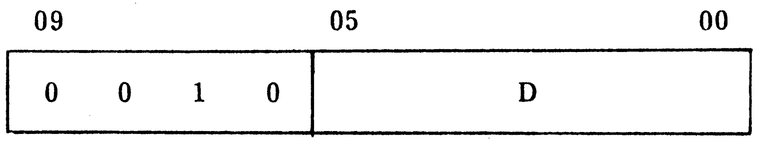
OP	Operation Code
M/D	Addressing Mode and Displacement



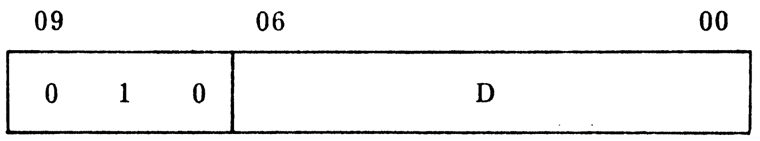
Scratchpad Byte: D



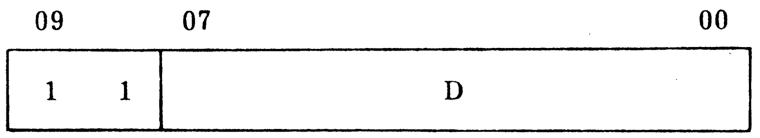
Relative Byte: (2P)+D-128



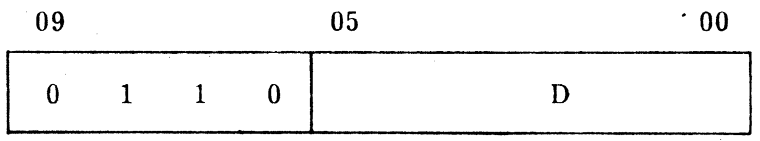
Indexed Byte: X+D



Indirect Scratchpad: *D



Indirect Relative: *(P+D-128)



Indirect Scratchpad
Post-Indexed: *D+X

MNEMONIC $\begin{bmatrix} * \\ @ \\ *@ \end{bmatrix}$ OPERAND

Prefixes:

*	Indirect Address
@	Indexed
*@	Indirect Post-Indexed

Skeleton	Mnemonic	Function
8800	ADDB	Add to A
9400	ANDB	AND to A
B800	CMSB	Compare A with Memory, Skip (Low, High, Equal)
9000	EMAB	Exchange Memory with A
B400	IORB	Inclusive OR to A
8000	LDAB	Load A
A000	LDXB	Load X
8400	STAB	Store A
A400	STXB	Store X
8C00	SUBB	Subtract from A
9800	XORB	Exclusive OR to A

ComputerAutomation

CUSTOMER INFORMATION BULLETIN

CIB No. 1228 - Known problems with RTX4 package Revision C1

1. RTX4

   a. It is not possible for the current activity to drop its own seniority to allow another activity of equal priority to resume. If R:PAUS is attempted the next (not the current) activity loses its seniority and is scheduled behind all ready-to-go tasks of the same priority as the current one.

   b. MDB:A macro generates an initial value of zero for the mailbox usage semaphore (instead of 1).

   c. FPMAX: no longer exists but is still described in the Manual (Page 5.3).

2. IOS4

   a. When reading from a VDU IOS4 will not check for backspacing beyond the beginning of a line.

   b. The SC (Skip lines) does not function for line printers.

   c. Top of Form is produced one line early for Centronics-type printers (i.e. with the auto-linefeed capability.

   d. If SB: is set when double-line spacing or top-of-form is carried out before a record is output the EOL or TOF sequence is output without the leading character.

   e. Unformatted Reads through PR and TY/TR turn parity off on incoming data.

   f. Formatted ASCII input does not detect embedded 'Rubout' characters (except at the beginning of a record).

3. SFM

   a. CREA:A macro defaults parameter 7 to zero instead of :7FFF.

CAI Limited
Technical Support Group

NOTES ON ITEMS ISSUED WITH RTX4 (C1)

1. RTX User's Manual (C0)

Appendix H describes the macro files (supplied with OS4 and RTX4 and their contents. The contents described for GEN.MAC should include all RTX4/IOS4/SFM service call macros.

2. IOS4 User's Manual (C0)

2.1 Similar comment as given in 1, except that it is Appendix G. Also page 8.1 refers to Appendix I instead of G and the Contents List has omitted the Appendix altogether.

2.2 Appendix B

The Introduction B.1 should include reference to the Volume Control Block and FLIST described later in the Appendix.

3. IOS4 (C1)

3.1 The IOS.HLP

This file includes description of the IOSDEMO program files. This demo is now called SFMDEMO.

3.2 The Line Printer DIB (Standard)

This is configured for 80 characters per line and 57 lines per page. The DIB:LP macro also defaults to these values and not 133 and 39 as described.

3.3 IOSD.MAC

Note that this file equates the CRT DIO channel address to 2 instead of 4 as one might expect.

NOTES ON ITEMS ISSUED WITH RTX4 (C1) (Cont.)

3.4 Write Direct Stream I/O

There is a fault connected with this. If a program attemps to do Write Direct Stream to a file in order to overwrite the exact number of bytes remaining in the file, SFM ignores the request and indicates an end of a block error (:4E). This fault may be overcome by patching as follows:

Location	Old Contents	New Contents
F:CEOF+:A	:9E82	:0000

The address of F:CEOF may be determined by examining the link-map produced by linking the user program with RTX/IOS/SFM.

3.5 TV/TK/TY End-of-Input Action

Currently, when carriage-return is required to terminate an input I/O request, IOS4 responds by repeating just that character, which means that it is possible for subsequent output to overprint the previously typed line. (In the case of OS4 message output, no overprinting occurs because a line-feed is output first, before the message.)

To ensure that no overprinting occurs, users may modify the location identified on link maps by the symbol TYELI:. Normally this location contains 1, but 2 should be put in its place to ensure that carriage -return is followed by a line-feed after every input line is terminated..

4. RTX (C1)

4.1 The fault described in connection with the previous version of RTX4 namely R:IWAL still exists and the same patch applies. For the benefit of those users new to RTX4, a copy of the EN issued just before this C1 release is attached to these notes.

NOTES ON ITEMS ISSUED WITH RTX4 (C1) (Cont.)

4.2 R:PAUS

This service should allow an activity to de-schedule itself so that it is placed at the end of the queued activities of the same prioity as itself. However, R:PAUS de-schedules the next activity in the queue. The following patch cures the fault:

Location	Old Contents	New Contents
R:PAUS+:8	:A022	:2922

4.3 MAILBOX

MDB:A macro is wrong. It allocates word containing 0 for Mailbox Usage Semaphore and it should contain 1.

Change source line 319 from "Word 0 - Mailbox Usage Semaphore" to "Word 1 - Mailbox Usage Semaphore".

4.4 RTX MACROS

TICK:A, WALL:A, MAIL:A,SDB:A,MDB:A Macros contain invalid constructions for testing n umber of parameters supplied with the call, e.g. 0<#?< 3.

There are no simple changes that can be made and users are advised to ensure that they provide the correct number of parameters since the macro definitions do not check correctly.

ComputerAutomation
18651 Von Karman, Irvine, Calif.

ENGINEERING NOTICE

NO. 7 1 6 9 S

DOCUMENT NO.	REV. IS	REV. WAS	TITLE	INCORP. DATE
093410-XX	B2	B1	RTX4 - R:IWAL	

EFFECTIVITY NOTES:

REASON FOR CHANGE:

REA NO. 04447

CO-ORD WITH:

CERTAIN COMBINATIONS OF R:TODL AND USER-SPECIFIED INTERVAL VALUES PRODUCE INCORRECT SHORT TIME INTERVALS BECAUSE THE CODE ASSUMES THAT THE ADD INSTRUCTION AFFECTS THE CARRY STATUS BIT, WHICH IT DOESN'T!

DESCRIPTION OF CHANGE:

A. PATCH AS FOLLOWS:

LOCATION	OLD CONTENTS	NEW CONTENTS
R:IWAL+:22	:C844	:0E07 RBIT 0,S
+:23	:C483	:4712) ADDC TL(Y),Q
+:24	:8843	:0004)
+:25	:56C1	:C483 COPY Q,CC:TL(X)
+:26	:0801	:0712) ADD~~C~~ TU(Y), A
+:27	:8482	:0003)
+:28	:9E80	:8482 COPY A,CC:TU(X)

(NOTE: JMP POST AT R:IWAL +:28 IS REDUNDANT BECAUSE POST IS THE NEXT LOCATION.)

TYPE	
AEN	☐
STOP ORDER	☐
DEVIATION	☐
RELEASE	☐
STANDARD	☒

CLASS	
A-MAND/FUNC	☒
B-NON-MAND/FUNC	☐
C-RECORD CHG	☐

AFFECTED ITEMS	PRIM.	SEC.
HARDWARE CHG.	☐	☐
SOFTWARE CHG.	☒	☐
PUBL. CHG.	☐	☐
CAPABLE CHG.	☐	☐
DOC. CHG.	☐	☐
CONFIGURATIONS	☐	☐
PROCEDURES	☐	☐
TOOLING	☐	☐
TEST EQUIP.	☐	☐

EFFECTIVITY ACTIVITY	U	R	S	P
NOTIFY VEND				
IN STOCK				
KITTING				
ASSEMBLY				
TOUCH UP				
IPT				
FIN GOODS				
CUST. RET.				

REWK TEST REQ'D	
CONTINUITY	☐
CABLE SCAN	☐
CAPABLE	☐
MEMORY	☐
CARD	☐
FINAL	☐
NO TEST REQ'D	☒

APPROVALS	
ENGR.	[illegible]
SOFTWARE	[illegible]
Q.A.	[illegible]
CAP. TEST	[illegible]
MAST SCHED	[illegible]
MATERIALS	[illegible]
TEST ENGR.	[illegible]
TECH SERV	[illegible]
CUST SERV	[illegible]
MFG. ENGR	[illegible]
PUBLICATIONS	[illegible]

DR. BY: A. DUZICH 12-
CHKD. BY: D. BURBECK 12-
REL BY: [illegible]